Universidad de Cádiz
Escuela Superior de Ingeniería

TRABAJO DE FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA

AUTOMATIZACIÓN DE TÉCNICAS DE PRUEBA DE SOFTWARE: IMPLEMENTACIÓN DE OPERADORES DE MUTACIÓN Y NUEVA FUNCIONALIDAD PARA MUCPP

AUTOR: MIGUEL ÁNGEL ÁLVAREZ GARCÍA

Puerto Real, septiembre 2019

Universidad de Cádiz
Escuela Superior de Ingeniería

TRABAJO DE FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA

AUTOMATIZACIÓN DE TÉCNICAS DE PRUEBA DE SOFTWARE: IMPLEMENTACIÓN DE OPERADORES DE MUTACIÓN Y NUEVA FUNCIONALIDAD PARA MUCPP

DIRECTOR: INMACULADA MEDINA BULO
CODIRECTOR: PEDRO DELGADO PÉREZ
AUTOR: MIGUEL ÁNGEL ÁLVAREZ GARCÍA

Puerto Real, septiembre 2019

Agradecimientos

A mi madre, por haber confiado en mí para realizar esta carrera, aunque las condiciones no fueran las óptimas para este desafío, por cuidarme y ayudarme en todo momento y ser la persona en la que confiar en todos y cada uno de mis días.

A mis abuelos, Antonio y Manoli, por ayudarme en todo lo que han podido, por confiar en mi en todo momento y ser una parte muy importante en este reto.

A mi tío, por cuidarme, por hacerme ser la persona que he llegado a ser, por ser brusco conmigo cuando me lo merecía, en fin, ser mi tipilititi.

A Inma, por su apoyo en todo momento, por su apuesta de futuro en mi y por su ayuda inconfundible cada vez que la he necesitado.

A Pedro, por su ayuda inconfundible para la puesta en marcha de este TFG, sus múltiples correos y correcciones, sin él, nada de esto podría haber salido adelante.

A la Delegación de Alumnos o Estudiantes de la ESI, mi Dele, mi segunda familia, por estar siempre ahí y haber contado conmigo durante estos cuatro años en numerosos proyectos. A todas esas personas que me llevo, muchísimas gracias.

A Juanjo, por su ayuda, su confianza en mí y su empuje hacia la finalización de este proyecto.

Al resto del centro, su dirección, su personal de administración y servicios, sus coordinadores de grado y sus directores de departamento con los que he tenido relación en algún momento, agradeceros todo lo que me habéis aportado en este camino.

Al resto del cuerpo universitario, su rectorado, su personal de administración y servicios, sus directores generales y de secretariado, a todos sus estamentos, a los que me hayan aportado una u otra cosa, sabiduría, saber relacionarme, hablar en público, organizar actos y eventos, gracias por haberme hecho ser lo que soy.

En definitiva, gracias a todos.

Licencia

Resumen

La Industria 4.0 está cada vez más presente dentro del tejido empresarial, transformando todos los aspectos relacionados con la producción. En este escenario, en el que las empresas se mueven dentro de un entorno completamente conectado y los sistemas software se vuelven cada vez más complejos, la prueba de software es un pilar fundamental a fin de medir y mejorar la calidad de los programas, suponiendo aproximadamente un 50 % del coste total del proyecto. Con las nuevas exigencias propias de la Industria 4.0, realizar las diferentes actividades implicadas en esta fase de forma manual y exhaustiva, podría suponer un coste incluso mayor, por lo que la automatización de pruebas sobre el software como la prueba de mutaciones son una gran ayuda para este fin, aportando la posibilidad de evaluar y mejorar las pruebas sobre el software. Es por ello que este proyecto busca la creación de nuevos operadores de mutación y mejoras en el funcionamiento de la herramienta de prueba de mutaciones MuCPP, una herramienta para el lenguaje C++, con el fin de su puesta en funcionamiento en la industria.

Palabras clave

Prueba de mutaciones

Operadores de mutación

MuCPP

Mutation tool

Prueba del software

Prueba de caja blanca

Desarrollo de software

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

El presente trabajo fin de grado ha sido desarrollado en colaboración con el Grupo UCASE [1] de Ingeniería del Software en su línea de investigación titulada "Prueba de software en la Industria 4.0", mediante la que se quiere conseguir la integración de su herramienta MuCPP en entornos industriales de la provincia.

1.1. Motivación

La Industria 4.0 está cada vez más presente dentro del tejido empresarial, transformando todos los aspectos relacionados con la producción. En este escenario, en el que las empresas se mueven dentro de un entorno completamente conectado, los sistemas software se vuelven cada vez más complejos.

La prueba de software es un pilar fundamental dentro del desarrollo de cualquier proyecto software a fin de medir y mejorar la calidad de los programas. Esta etapa de pruebas puede suponer sobre un 50 % del coste total de un proyecto [2], tanto económico como en tiempo, aunque en entornos de carácter industrial esta etapa puede bajar al 40 %. Con las nuevas exigencias propias de la Industria 4.0, realizar las diferentes actividades implicadas en esta fase de forma manual y exhaustiva podría suponer un coste incluso mayor. Además, la presencia de errores en el software no detectados antes de su puesta en ejecución puede tener graves consecuencias, especialmente en sistemas críticos.

Dado este contexto, se hace necesaria una evolución hacia la automatización de la prueba de software, incorporando técnicas que permitan incrementar la confianza en que nuestro sistema carece de fallos. Una de las técnicas más potentes en este sentido es la prueba de mutaciones [3], que consiste en introducir pequeños fallos en el código fuente de un programa dado y observar si dicho cambio es identificado con las pruebas realizadas. Estos fallos se inyectan en base a unas reglas de transformación predefinidas, que simulan fallos de programación habituales, y que se conocen como operadores de mutación.

Solo con la mejora de las herramientas para que se ajusten a las nuevas necesidades de la industria se conseguirá la necesaria acogida e integración de técnicas avanzadas de prueba, como la prueba de mutaciones, en procesos de prueba reales.

Dado que la investigación en torno a esta técnica es continua desde hace años, aportando entre otros aspectos nuevos operadores de mutación, herramientas para diversos lenguajes y formas de reducir el coste, se hace necesario un seguimiento de estos avances y de la actualización de las herramientas para mejorar las prestaciones de las mismas en la práctica.

1.2. Objetivos

En la actualidad, el lenguaje de programación más extendido en la industria es C++, un lenguaje de programación diseñado en 1979, y utilizado por primera vez en 1983 fuera de un laboratorio científico. Para este lenguaje existen muy pocas herramientas para la prueba de mutaciones, entre las que se encuentra una herramienta denominada MuCPP, desarrollada en el seno de la Universidad de Cádiz, concretamente por miembros del Grupo de investigación UCASE de Ingeniería del Software (TIC025) [1].

MuCPP incluye diversos operadores de mutación, en concreto, un conjunto de operadores de mutación de los denominados "tradicionales" y otro conjunto a nivel de clase (respecto de la parte orientada a objetos de dicho lenguaje).

A pesar de que de la herramienta se ha mostrado efectiva en su aplicación a varios proyectos [4], se observa una necesidad de actualización y extensión de la misma para su mejora y facilitar su uso en la industria. Entre las mejoras se encuentran la actualización del compilador que se usa de forma subyacente dentro de la herramienta para poder analizar el código y detectar posibles puntos donde mutar, Clang, que se encuentra en su versión 8 en la actualidad y en su versión 3.6 en la herramienta, suponiendo no poder usar la herramienta en sistemas con otra versión del compilador. Por otro lado, encontramos la actualización del propio lenguaje, C++, el cual ha pasado desde su versión 2003 a la actual versión de 2017, aportando diferentes mejoras de las que no se crean mutantes dado que no se han incluido nuevos operadores desde esa versión. Destacar también la dificultad del uso de esta herramienta en código fuente de gran extensión debido a que su uso implica la creación de los mutantes del código completo.

Por todo ello, el presente proyecto lleva a cabo una actualización e implementación de nuevos operadores de mutación, una actualización del compilador y una actualización a nivel de funcionamiento, en concreto se realizarán las siguientes mejoras y actualizaciones:

- Actualización e implementación de nuevos operadores aparecidos en la literatura y en diferentes herramientas de prueba de mutaciones.
- Actualización de Clang de su versión 3.6 a su versión 8.
- Actualización a nivel de funcionamiento para una fácil inclusión de la herramienta en la Industria actual, donde poder expresar la clase o función específica de la que queremos crear los diferentes mutantes.

Todo ello, para conseguir una herramienta totalmente actualizada y poder ser utilizada en la Industria, facilitando su utilización de una forma más flexible y efectiva.

1.3. Descripción general del proyecto

El presente proyecto se centra en la prueba de mutaciones, una técnica que consiste en la introducción de pequeños fallos en el código fuente de un determinado programa y observar si dicho cambio es identificado o no por las pruebas realizadas por el desarrollador. Estos fallos son inyectados en base a reglas de transformación predefinidas que simulan fallos de programación habituales, que son denominados *operadores de mutación*.

La prueba de mutaciones es una técnica de caja blanca, siendo esta una prueba del software que consiste en escoger distintos valores de entrada para un programa del que se conoce su código fuente examinando todos y cada uno de sus posibles flujos de ejecución. Estas pruebas son aplicables a diferentes niveles como pueden ser clases o métodos.

En el proyecto se implementarán diferentes operadores de mutación a la herramienta MuCPP, la cual, está basada en el lenguaje C++. Este lenguaje tiene múltiples versiones y estándares, siendo el estándar más conocido el ISO/IEC C++, el cual inicialmente especifica los requisitos para las implementaciones con C++98, C++03 y C++11, aportando posteriormente diferentes versiones cada tres años, 2014, 2017 y 2020.

En este punto también es importante destacar que MuCPP es una herramienta creada con Clang, uno de los múltiples subproyectos de LLVM, el cual, es un compilador *frontend* de código abierto para los lenguajes de programación C, C++ y Objetive-C.

LLVM es un proyecto que tiene como finalidad la creación de nuevos compiladores optimizados para cualquier lenguaje de programación, suministrando la infraestructura necesaria para el desarrollo de compiladores actuando como *backend* al tomar el código intermedio que los distintos *frontends* generan para los diversos lenguajes. Gracias a estas tecnologías es posible la implementación de los operadores para generar los mutantes de una manera muy robusta, y en este proyecto además permitirá delimitar la zona en la que un usuario quiere realizar esas mutaciones, es por ello que las bibliotecas de Clang que usa MuCPP deben estar lo más actualizadas posibles, teniendo además en cuenta que el propio lenguaje C++ está en constante evolución.

1.4. Alcance

La herramienta actualizada al final del presente Proyecto Fin de Grado pretende ser empleada en la Industria. Además, tendrá un uso científico, tanto en docencia como en investigaciones que pueda surgir sobre la prueba de mutaciones y que será de gran aportación para la literatura existente sobre este tipo de pruebas.

Para la utilización de la presente actualización será necesaria un ordenador con el sistema operativo Ubuntu 18.04 LTS además de software de fácil instalación.

A la finalización de este proyecto se habrá generado lo siguiente:

- Un estudio sobre los operadores aparecidos en la literatura y en diferentes herramientas de prueba de mutaciones.
- Conjunto de ficheros de código fuente con la implementación de los nuevos operadores de mutación contemplados en el estudio.

- Diferentes ficheros con código fuente para probar los diferentes operadores operadores indicados anteriormente.
- Actualización de la herramienta MuCPP con los operadores descritos anteriormente, además de una actualización interna para su uso en una parte restringida del código completo.

1.5. Organización del documento

El presente documento se estructura en una serie de capítulos, a través de los se describirán detalladamente todas y cada una de las etapas por las que ha ido pasando el presente proyecto desde sus orígenes hasta su finalización. A continuación se realiza un breve resumen de lo que se tratará en cada capítulo.

- **Capítulo 1.** En este capítulo se realizará una introducción al proyecto, pasando por la motivación para su realización, los objetivos, una descripción general del proyecto y el alcance que se tendrá, además de esta sección en la que se explica la estructura del documento.
- **Capítulo 2.** En este capítulo se contará con la planificación que se ha tenido en la realización del presente proyecto, cuantificando tanto en tiempo como en dinero el coste general del proyecto.
- **Capítulo 3.** En el capítulo tres se realizará una introducción a los conceptos generales. Entre estos conceptos se encuentran el lenguaje C++, las pruebas del software, centrándose en la de mutaciones, el proyecto LLVM, Clang y el árbol de sintaxis abstracta o AST. Por último se verá el estado del arte, describiendo las principales herramientas de prueba de mutaciones que se pueden encontrar en la actualidad.
- **Capítulo 4.** En el capítulo cuatro se realizará una breve introducción a los operadores de mutación que se pueden encontrar en las diferentes herramientas de prueba de mutaciones con lo que se podrá observar las diferencias y similitudes entre herramientas.
- **Capítulo 5.** En este capítulo se realizará una comparativa sobre los operadores que ya se encuentran en la herramienta MuCPP con respecto a otras herramientas.
- **Capítulo 6.** En el capítulo seis se realizará una extensa explicación sobre los nuevos operadores que se han implementado en la herramienta MuCPP, basándose en diversos artículos de investigación y otras herramientas.
- **Capítulo 7.** Se explicará las mejoras que se han implementado en la herramienta de pruebas MuCPP, pasando por el hacer que la herramienta solo cree mutantes para una clase, función o método concreto y las dos actualizaciones realizadas en torno al proyecto sobre el que se basa la herramienta, Clang.
- **Capítulo 8.** En este capítulo se describen las pruebas realizadas a todos y cada uno de los nuevos operadores implementados para la herramienta MuCPP.
- **Capítulo 9.** En este capítulo se explicarán las conclusiones obtenidas mediante la realización de este proyecto y el trabajo fututo a realizar.

- **Anexo A.** En este anexo se realiza una explicación sobre la instalación de la herramienta, además de un manual de uso de esta.
- **Anexo B.** En el anexo B se realiza una breve explicación de como instalar los diferentes operadores individuales que se han aportado como adjunto al presente documento, además de como se utilizarán para obtener los diferentes mutantes que genera cada operador.

Capítulo 2

Planificación

2.1. Planificación del tiempo

El presente proyecto se ha realizado conforme a cuatro meses de trabajo, dividiéndose este en diferentes cursos académicos. Destacar que todas las fases aquí expuestas son desarrolladas durante el transcurso del presente documento.

2.1.1. Fase 1: Investigación sobre la prueba de mutaciones

En este periodo de tiempo se ha investigado sobre la prueba de mutaciones, diferentes herramientas, lenguajes en los que se ha desarrollado más y los últimos operadores expuestos en la literatura.

2.1.2. Fase 2: Conocimiento del software a mejorar

En esta etapa se conoce el software que se desea mejorar en etapas posteriores. Se considera una etapa de gran utilidad dado que se van creando diferentes operadores de mutación de fácil realización para coger práctica, además de observar y comprender el código de MuCPP y cómo realiza sus acciones.

2.1.3. Fase 3: Desarrollo de operadores de mutación

Tras la etapa anterior, se crean los operadores de mutación que no se encontraban en la herramienta MuCPP pero sí en otras, además de actualizar los diferentes operadores que se han considerado que no tenían las mismas características que los de otras herramientas.

2.1.4. Fase 4: Desarrollo de mejoras para MuCPP

En esta etapa se realizan las mejoras de la herramienta MuCPP, empezando por la actualización de Clang, la cual dura diferentes semanas dada la complejidad que supone cambiar entre versiones muy lejanas. Posteriormente se inserta la mejora de realización de las pruebas de mutación en una determinada clase o función/método, la cual debe obtener el código que se inserta por consola, analizarlo y pasarlo como parámetro al AST Matcher de Clang previamente actualizado.

2.1.5. Fase 5: Pruebas sobre el software

Etapa en la que se prueban todos y cada uno de los operadores de mutación actualizados e implementados con el fin de asegurar su correcto funcionamiento antes de ser entregados. Cabe destacar que mientras se encontraban en desarrollo han sido probados con multitud de entradas y cambios, obteniendo así los operadores que se buscaban.

2.1.6. Fase 6: Documentación

En esta etapa se termina de desarrollar el presente documento, ya que ha sido desarrollado durante todas las etapas anteriores de forma paralela.

2.1.7. Diagrama de Gantt

En la presente sección se muestra el diagrama de Gantt con el transcurso completo del proyecto.

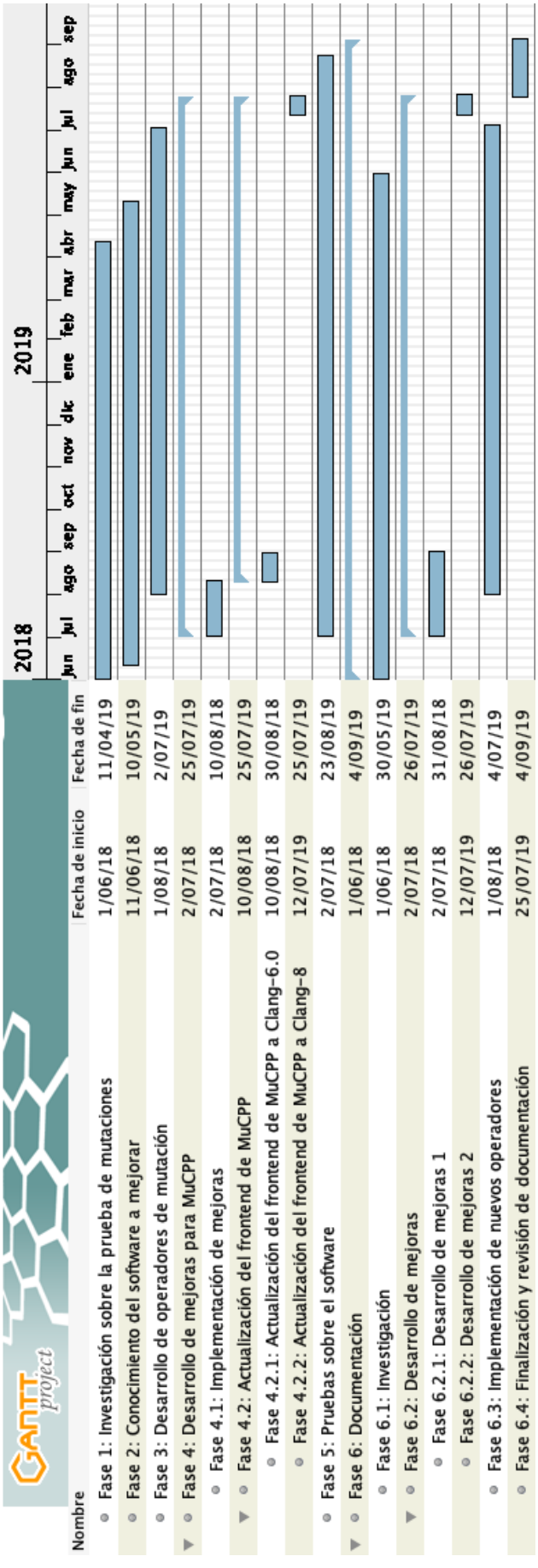

Figura 1: Diagrama de Gantt.

2.2. Coste general del proyecto

En esta sección se desarrollará un presupuesto del coste total aproximado del proyecto realizado, indicando las horas trabajadas, de investigación, de conocimiento del software, de desarrollo y de documentación. En la tabla 2.2 pueden observarse las horas empleadas para cada fin, además de los costes totales de cada parte del proyecto. Destacar que para el calculo del coste/hora se toma teniendo en cuenta el de un único trabajador.

Detalle	Horas	Coste/Hora	Coste total
Investigación sobre la prueba de mutaciones y los últimos operadores aparecidos.	150	22 €	3.300 €
Conocimiento del software a mejorar.	75	22 €	1.650 €
Desarrollo de operadores de mutación.	225	22 €	4.950 €
Desarrollo de mejoras para MuCPP.	175	22 €	3.850 €
Pruebas sobre el software.	85	22 €	1.870 €
Documentación.	100	22 €	2.200 €
Coste total del proyecto.	725	22 €	**17.820€**

Tabla 1: Tabla coste general del proyecto.

Para el cálculo total de horas se ha llevado realizado una estimación en cuento a las horas de trabajo en cada ámbito del proyecto. A continuación se detallan cada uno de los aspectos contenidos en la tabla.

- **Investigación sobre la prueba de mutaciones y los últimos operadores aparecidos**. Se estiman un total de ciento cincuenta horas en este aspecto, considerando que se ha tenido que conocer la prueba de mutaciones, investigar sobre otras herramientas, conocer qué son los operadores de mutación y cómo funcionan, entre otros.
- **Conocimiento del software a mejorar**. Se estiman un total de setenta y cinco horas en este aspecto, considerando que se ha tenido que conocer la herramienta al completo, los operadores de mutación que poseía y cómo funcionaba internamente.
- **Desarrollo de operadores de mutación**. Se estiman un total de doscientas veinticinco horas en este aspecto, considerando que se ha tenido que actualizar los operadores que se encontraban en MuCPP además de desarrollar nuevos operadores.
- **Desarrollo de mejoras para MuCPP**. Se estiman un total de ciento setenta y cinco horas considerando que se ha tenido que incluir nuevas funcionalidades a la herramienta además de actualizar en dos ocasiones la versión de Clang.
- **Pruebas sobre el software**. Se estiman un total de ochenta y cinco horas en este aspecto ya que se ha tenido que realizar diferentes pruebas para los nuevos operadores desarrollados las cuales se adjuntan en este documento.
- **Documentación**. Se estiman un total de cien horas en este aspecto, considerando que se han tenido que escribir una cantidad considerable de páginas además de realizar diferentes revisiones de estas.

Capítulo 3

Conceptos y estado del arte

3.1. El lenguaje C++

C++ [5] es un lenguaje de programación de propósito general diseñado con la intención de extender al lenguaje C con la inclusión del paradigma de la Orientación a Objetos y fue diseñado en 1979 por Bjarne Stroustrup.

Por tanto, este lenguaje es orientado a objetos, aunque no "puro", dado que soporta otros estilos de programación como el estructurado. Por ello, también se le denomina lenguaje híbrido. En palabras de su creador, el hecho de que C++ no sea un lenguaje de orientación a objetos puro es más una ventaja que un inconveniente, ya que lo hace más versátil y adecuado para un mayor número de aplicaciones. Más tarde se añadieron facilidades de programación genérica, que, junto a los paradigmas de programación estructurada y programación orientada a objetos, se suele decir que c++ es un lenguaje de programación multiparadigma.

Al igual que ocurre con otros lenguajes como C, el lenguaje C++ fue normalizado por el comité X3J16 de ANSI, el cual, se constituyó en 1989 y terminó su trabajo en 1997. La versión normalizada se denominó C++ ANSI.

Posteriormente se creó el estándar ISO/IEC C++, mediante el comité *ISO/IEC 14882:2011 Information technology - Programming languages - C++*, publicándolo en Génova en Febrero de 2012. Este estándar especifica los requisitos para la implementaciones del lenguaje de programación C++ con sus primeras versiones C++98, C++03 y C++11. Actualmente este estándar está aportando versiones cada tres años, en concreto, C++14, C++17 y C++20 a las cuales se han adherido la mayoría de fabricantes de compiladores más modernos.

Además de las facilidades proporcionadas por C, C++ proporciona otras muchos conceptos, como tipos de datos adicionales, clases, plantillas, excepciones, espacios de nombres, sobrecarga de operadores, sobrecarga de nombres de funciones, referencias e instalaciones de bibliotecas adicionales. Podemos determinar que el lenguaje C++ ha evolucionado hasta llegar a ser un lenguaje de propósito general ligeramente recortado.

La denominación C++ fue propuesta por Rick Mascitti en el año 1983, cuando el

lenguaje fue utilizado por primera vez fuera de un laboratorio científico. Antes de este momento se le había denominado como "C con Clases" [6].

C++ es, a día de hoy, uno de los lenguajes más empleados, tal y como recoge su cuarta posición en el ranking de lenguajes TIOBE [7], por detrás de Java, C y Python. Este lenguaje es usado especialmente en la industria de las comunicaciones, aeronáutica, militar, y un largo etcétera.

3.2. La pruebas del software

Una de las fases más importantes y complejas en el desarrollo de un software es la denominada como prueba del software [8, 9], la cual es un "filtro" para todo el desarrollo. Este proceso sirve para descubrir errores y defectos a fin de poder eliminarlos.

Existen diferentes definiciones para el concepto de pruebas del software, entre los que se encuentra el definido por Glenford J. Myers [10]:

> ***Las pruebas son el proceso de ejecución de un programa con la intención de encontrar errores.***

No es posible asegurar que un software es totalmente correcto, ya que, no es posible realizar una prueba exhaustiva analizando todas y cada una de las posibles situaciones a las que el software se va a enfrentar en el futuro.

Los fallos pueden ser de diferentes tipos como errores de precisión, en las prestaciones, en la documentación o incluso en los procedimientos seguidos durante el desarrollo que dificultarán el mantenimiento del sistema. Una estrategia de prueba de software proporciona una guía que describe los pasos a realizar como parte de dicha prueba, además del esfuerzo, tiempo y recursos que requerirá [8, 9].

Existen múltiples clasificaciones de pruebas del software, entre las que encontramos las pruebas por su ejecución, según lo que verifican o por su enfoque, las cuales tienen diferentes pruebas, como son:

- Por su ejecución:
  - **Pruebas manuales.** Aquellas que son ejecutadas por una o mas personas simulando las acciones de un usuario final.
  - **Pruebas automáticas.** Pruebas que son ejecutadas mediante scripts para que sean realizadas más rápidamente.
- Por lo que verifican:
  - **Pruebas funcionales.** Pruebas diseñadas para comprobar las funcionalidades previamente diseñadas.
  - **Pruebas no funcionales.** Toda prueba que no compruebe la función que realiza el software, por ejemplo, usabilidad, portabilidad o eficiencia.

- Por su enfoque:
  - **Pruebas de caja blanca.** Pruebas que se basan en el acceso al código fuente, comprobando el acceso a todos los escenarios posibles.
  - **Pruebas de caja negra.** Pruebas que no tienen acceso al código fuente. Se basan en la introducción de diferentes entradas, comprobando que sus salidas son las esperadas.

A continuación se describirán las clasificaciones más utilizadas, como son las pruebas funcionales y las no funcionales o las pruebas de caja blanca y pruebas de caja negra.

3.2.1. Las pruebas funcionales del software

Entre las pruebas funcionales del software encontramos aquellas pruebas que se basan en la ejecución, revisión y retroalimentación de las funcionalidades del software en cuestión. Entre estas pruebas encontramos distintos tipos [9]:

- **Pruebas unitarias.** Son aquellas pruebas que dirigen los esfuerzos de verificación a la unidad más pequeña del diseño del software, centrándose en detectar errores de datos, lógica o algoritmos.
- **Pruebas de integración.** Las pruebas de integración son aquellas pruebas que se realizan para comprobar que los elementos unitarios se comportan de manera adecuada en conjunto. Existen dos tipos de pruebas de integración:
  - **Incremental.** Las pruebas se llevan a cabo probando cada módulo con el conjunto de módulos que ya haya sido probado. En este caso, existen diferentes estrategias, como son: la integración descendente, la integración ascendente, la prueba de regresión o la prueba de humo.
  - **No incremental.** Las pruebas se llevan a cabo probando cada módulo de forma independiente y, posteriormente, se combinan todos para formar el programa completo.
- **Pruebas alpha.** Pruebas realizadas mientras el software se encuentra en desarrollo y que tienen como fin comprobar que el desarrollo se está realizando de manera correcta y satisfaciendo las necesidades del cliente.
- **Pruebas beta.** Pruebas realizadas tras las pruebas alpha y la finalización del desarrollo, haciéndolo en un entorno real y cuyo fin es el de detectar fallos no vistos anteriormente.
- **Pruebas de aceptación.** Pruebas a realizar antes de la liberación de nuevas versiones, comprobando que el software cumple con las expectativas del usuario final.
- **Pruebas de regresión.** Pruebas realizadas con el fin de asegurar que los casos de prueba que ya habían sido probados y fueron exitosos permanezcan así tras una actualización del software.
- **Pruebas de sistema.** Pruebas que se realizan a nivel global tras las pruebas unitarias y las pruebas de integración explicadas anteriormente.

- **Pruebas de humo.** Pruebas que tienen como fin el evaluar la calidad del software desarrollado, en el que se realizan pruebas del funcionamiento básico antes de ser entregado al usuario final.

3.2.2. Las pruebas no funcionales del software

Entre las pruebas no funcionales del software encontramos aquellas pruebas cuyo objetivo es la verificación de un requisito que especifica criterios no funcionales como la disponibilidad, accesibilidad, usabilidad, mantenibilidad, seguridad, rendimiento, etc. Podemos clasificar las pruebas no funcionales según el tipo de requisito no funcional que prueba:

- **Pruebas de compatibilidad.** Son aquellas pruebas que verifican el funcionamiento de un determinado software en diferentes entornos.

- **Pruebas de seguridad.** Pruebas que miden la seguridad de un determinado software, estas pruebas pueden ser realizadas antes de su puesta en funcionamiento por el usuario final o posteriormente.

- **Pruebas de stress.** Pruebas que miden el nivel de esfuerzo que se le puede aplicar a un determinado software, determinando así la carga de trabajo que soportará.

- **Pruebas de usabilidad.** Es una prueba centrada en el usuario mediante la cual se pueden observar las interacciones que estos realizan en el software con el fin de mejorarlas.

- **Pruebas de rendimiento.** Son aquellas pruebas en las que se mide el tiempo que tarda el software en realizar determinadas tareas en un entorno.

- **Pruebas de escalabilidad.** Pruebas que se realizan para comprobar el nivel de escalabilidad que posee un determinado software, o en otras palabras, cuanta demanda puede soportar sin realizar grandes cambios en su configuración.

- **Pruebas de mantenibilidad.** Pruebas que se realizan con el fin de mantener el software, realizando, por ejemplo, limpiezas de datos almacenados y que no serán utilizados en el futuro.

- **Pruebas de instalabilidad.** Son pruebas que se realizan en diferentes entornos con el fin de comprobar la dificultad de su instalación.

- **Pruebas de portabilidad.** Pruebas realizadas para observar cuan portable es un software, es decir, cambiarlo de una máquina a otra que posea el mismo entorno.

3.2.3. Las pruebas de caja blanca

Las pruebas de caja blanca son aquellas que se centran en los procedimientos del software por lo que su diseño está fuertemente ligado al código fuente. Para ello, se escogen distintos valores de entrada para examinar cada uno de los posibles flujos de ejecución del programa y garantizar que se devuelven los valores de salida adecuados. Al estar basadas en una implementación concreta, si esta se modifica, por regla general las pruebas también deberán rediseñarse.

Aunque las pruebas de caja blanca son aplicables a varios niveles, habitualmente se aplican a las unidades de software, en el que se comprobará todos los flujos de ejecución dentro de cada unidad y entre unidades.

3.2.4. Las pruebas de caja negra

Las pruebas de caja negra son aquellas pruebas en las que se el software se utiliza de manera aislada y sin conocer su código fuente, solo conociendo sus entradas y las salidas que se producen, sin tener en cuenta su funcionamiento interno.

En otras palabras, en las pruebas de caja negra interesa su forma de interactuar con el medio que le rodea, entendiendo qué es lo que hace, pero sin dar importancia a cómo lo hace. Por tanto, en estas pruebas deben estar bien definidas sus entradas y salidas. En cambio, no se precisa definir ni conocer los detalles internos de su funcionamiento.

3.2.5. La prueba de mutaciones

La prueba de mutaciones es una técnica de caja blanca que consiste en la introducción de pequeños fallos en el código fuente de un determinado programa y observar si dicho cambio es identificado o no por las pruebas realizadas por el desarrollador. Estos fallos son inyectados en base a reglas de transformación predefinidas que simulan fallos de programación habituales, que son denominados *operadores de mutación*.

Para conocer la historia de la prueba de mutaciones debemos remontarnos a 1971, donde un estudiante llamado Richard Lipton dio esta idea en uno de sus artículos [11], pero encontró muchos problemas relacionados con su viabilidad de puesta en marcha en aplicaciones prácticas. Posteriormente, a finales de los años 70, se publicaron los principales artículos sobre este tema con Hamlet en 1977 [12] y con DeMillo en 1978 [13], donde introdujeron formalmente la mutación como una técnica de prueba en sus artículos. Recientemente los avances en la investigación de este tema han acercado a la realidad un sistema práctico de prueba de mutaciones.

En este campo, es denominado *mutante* al código fuente completo, con la inserción de una mutación mediante un operador de mutación específico. A modo de ejemplo se puede considerar el siguiente mutante, siendo el código fuente idéntico al original con el cambio *i=i+1*, cuando en el programa original se contemplaba como *i=i-1*. Una vez obtenido el mutante, este es ejecutado sobre el conjunto de casos de prueba realizados por el desarrollador, pudiendo obtener diferentes salidas al programa original, las cuales pueden ser observadas en la Figura 2.

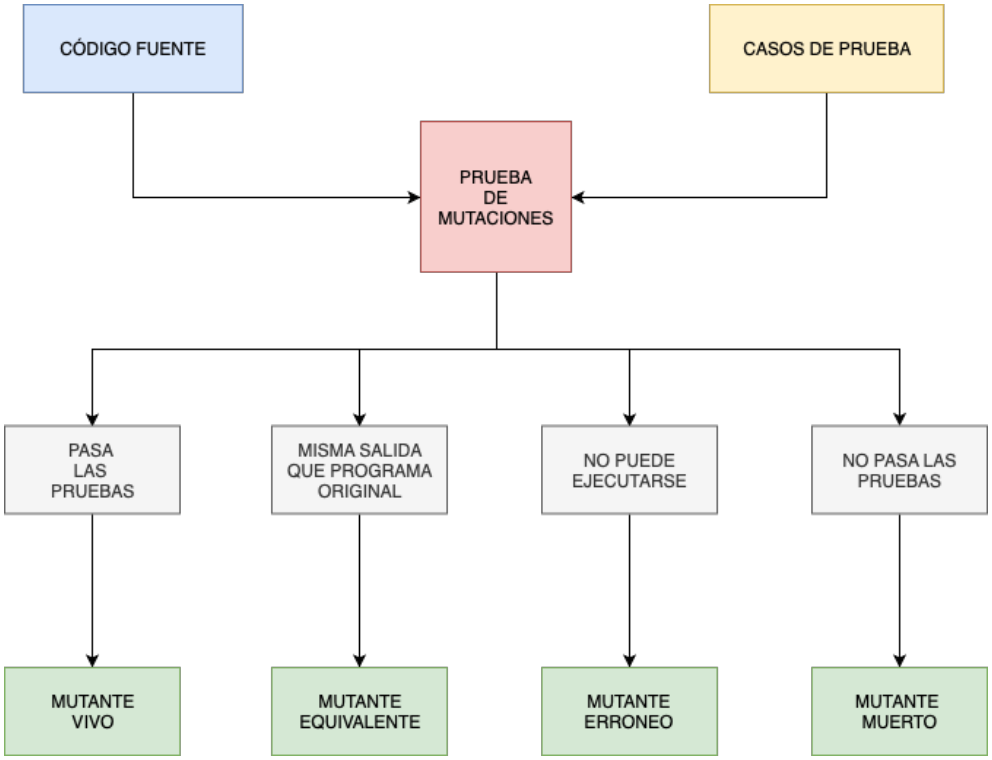

Figura 2: Tipos de salidas en la prueba de mutaciones.

Tal y como se ha podido observar, la prueba de mutaciones tiene como entrada el código fuente de un programa y el conjunto de casos de prueba, pudiendo aportar diferentes salidas.

- **Mutante vivo**. El mutante que pasa las pruebas aportadas se considera mutante vivo y significa que las pruebas deben ser mejoradas.
- **Mutante equivalente**. El mutante que da la misma salida que el programa original se considera mutante equivalente.
- **Mutante erróneo**. Aquel mutante que no puede ejecutarse se denomina mutante erróneo.
- **Mutante muerto**. Aquellos mutantes que no pasan las pruebas realizadas por el desarrollador mueren, y tiene como significado que las pruebas realizadas son efectivas con ese mutante.

Un ejemplo de mutante puede ser aquel cuyo operador de mutación es el intercambio de un operador relacional. Por ejemplo, dado un programa P que contiene una instrucción z<10, pueden ser generados diferentes mutantes entre los que se encontrará, por ejemplo, x>10.

Por otro lado, encontramos la forma de medir la calidad de un conjunto de casos de prueba mediante la denominada *puntuación de mutación* (mutation score). Esta puntuación nos indica el porcentaje de mutantes muertos frente al de mutantes no equivalentes. Este porcentaje se calcula de la siguiente forma:

$$PM\ (P,\ C) = \frac{MM}{TM - ME} \cdot 100$$

Siendo:
***PM**: Puntuación de mutación*
***P**: Programa testeado*
***C**: Conjunto de casos de prueba*
***MM**: Mutantes muertos*
***TM**: Total de mutantes*
***ME**: Mutantes equivalentes*

3.3. El proyecto LLVM y Clang

3.3.1. LLVM

LLVM [14] es un proyecto que tiene como finalidad la creación de nuevos compiladores optimizados para cualquier lenguaje de programación. Para ello, LLVM suministra la infraestructura necesaria para el desarrollo de compiladores, actuando como backend al tomar el código intermedio que los distintos frontends generan. Se trata de un proyecto de código abierto que engloba un gran número de subproyectos para la creación de nuevos frontends que trabajen sobre LLVM, muchos de los cuales son empleados en otros proyectos, tanto comerciales como de código abierto, así como en el ámbito académico [15].

Para comprender el origen de Clang [16], es necesario conocer la existencia del proyecto LLVM, el cual, comenzó como una investigación en la Universidad de Illinois con el objetivo de proporcionar una estrategia de compilación basada en la SSA capaz de soportar tanto la compilación estática como la dinámica [14].

SSA, abreviación de asignación estática única o *static single assignment*, es una propiedad de la representación intermedia de los programas o IR diseñadas para permitir diversas optimizaciones, en la que se requiere que cada variable se asigne únicamente una vez y sea definida antes de usarse, permitiendo al compilador hacer una asignación de registro más eficiente.

El subproyecto de LLVM que se contempla en el presente Proyecto tiene el nombre de Clang, un compilador de código abierto, que proporciona el acceso al AST o Arbol de Sintaxis Abstracta, mediante el cual podremos conseguir el objetivo de la prueba de mutaciones, la inserción de pequeñas modificaciones en el código fuente.

3.3.2. Clang

Clang es un compilador *frontend* de codigo abierto para los lenguajes de programación C, C++ y Objective-C, el cual usa LLVM en su *backend* y cuyo objetivo es una compilación rápida y proporcionar información eficiente sobre errores y advertencias.

Tal y como se mencionó en el apartado anterior, LLVM proporciona la forma intermedia del proceso de compilación mediante un AST, el cual contiene la estructura completa

del código fuente que se desea analizar, mediante ramas del árbol, facilitando así el acceso a partes del código. Además, Clang mantiene el código escrito por el desarrollador, cosa que no ocurre en otros compiladores como GCC, los cuales simplifican el código. Esto es beneficioso a la hora de crear mutantes, dado que la intención es conservar el mismo código escrito por el programador e insertarle diferentes modificaciones mediante los diferentes operadores de mutación.

Mediante la combinación de LLVM y Clang se obtiene un importante conjunto de herramientas que permiten el desarrollo de nuevas herramientas como es el caso de MuCPP [17], del cual hablaremos posteriormente.

3.4. Arbol de sintaxis abstracta o AST

Un árbol de sintaxis abstracta o AST es una representación estructurada de un determinado código tras sus correspondientes análisis léxicos, sintáctico y semántico. Las expresiones del código están expresadas mediante la combinación de diferentes ramas del árbol, lo cual nos aporta una visión más clara de la estructura del código que nos facilita la búsqueda de aquellos nodos que cumplen los criterios de los diferentes operadores de mutación.

El AST generado por Clang posee un formato parecido a XML, lo cual lo hace más comprensible incluso por aquellos que aún no están familiarizados por la forma en la que un compilador funciona internamente. Para cada token o nodo, Clang guarda información de en que parte del código fue escrito o donde fue expandido si se trata de una macro.

El AST de Clang es posible extraerlo para poder observar su forma y analizar el código de un fichero de código fuente dado. Esto es posible mediante la siguiente orden ejecutada en una terminal sobre el fichero *ejemplo.cpp*:

```
clang++ -Xclang -ast-dump -fsyntax-only ejemplo.cpp
```

A continuación se expresa un ejemplo del funcionamiento de este AST, haciendo uso de un programa que contiene una única sentencia que es: *a=1+2+3;*. De esta manera, observamos el siguiente AST:

```
-CompoundStmt 0x55b9ea06c1e0 <col:39, line:6:1>
    |-DeclStmt 0x55b9ea06c190 <line:4:2, col:13>
    | `-VarDecl 0x55b9ea06c080 <col:2, col:12> col:6 a 'int' cinit
    |   `-BinaryOperator 0x55b9ea06c168 <col:8, col:12> 'int' '+'
    |     |-BinaryOperator 0x55b9ea06c120 <col:8, col:10> 'int' '+'
    |     |  |-IntegerLiteral 0x55b9ea06c0e0 <col:8> 'int' 1
    |     |  `-IntegerLiteral 0x55b9ea06c100 <col:10> 'int' 2
    |     `-IntegerLiteral 0x55b9ea06c148 <col:12> 'int' 3
```

Tal y como podemos observar, el nivel de sangría o identación que se obtiene en cada una de las sentencias indica la profundidad de un elemento del árbol.

Como se puede ver, en primer lugar se realizan las diferentes operaciones binarias representadas por la clase *BinaryOperator* en la API de Clang, en este caso, primero se realizará

la operación *1+2* y posteriormente se le suma el *3*, dado que en primer lugar se realizan las operaciones más internas y posteriormente las más externas. Una vez realizado este cálculo, se declara la variable *a* como entera y se le asigna el valor calculado anteriormente.

Destacar también que estas sentencias son únicamente una parte del árbol generado, siendo la única que hace referencia a la sentencia que nos interesa en este caso, *a=1+2+3;*.

A continuación se puede observar el árbol generado de una forma más visual a la anterior:

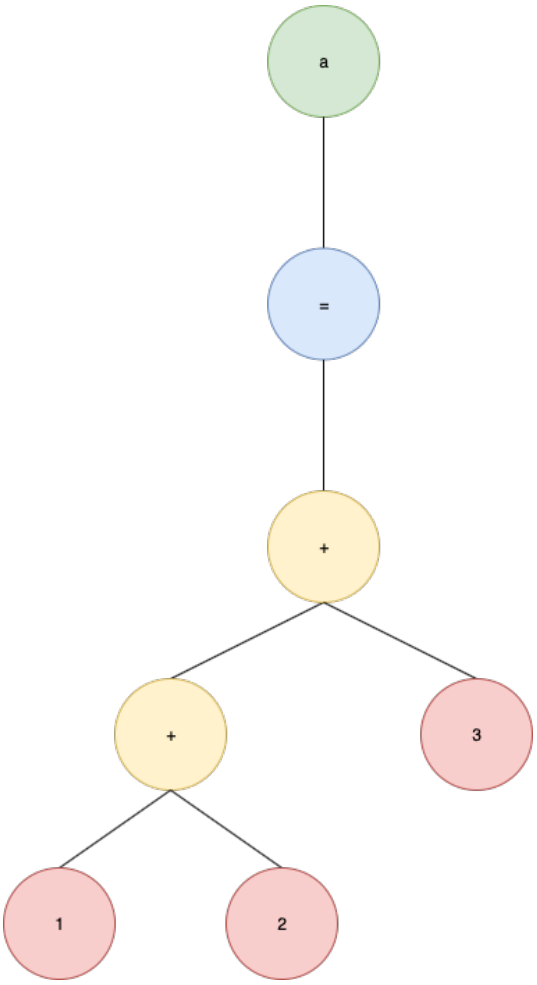

Figura 3: Arbol ejemplo AST.

3.5. Estado del arte

La prueba de mutaciones en sus primeros años de desarrollo se basó en lenguajes estructurados, dado que en esa época aún se encontraban en desarrollo los lenguajes orientados a objetos. Por ello, podemos encontrar herramientas para lenguajes como Ada, Fortran o C.

En la actualidad existen múltiples herramientas para lenguajes como Java o Python, pero en cambio, son mínimas las creadas para el lenguaje C++ y aquellas que se han encontrado son privativas y aportan poca o ninguna información. De hecho, ni siquiera se documentan los operadores de mutación que se pueden encontrar en su interior, por lo que no son de utilidad para el presente proyecto.

En el caso de los sistemas de mutaciones para el lenguaje que más nos importa, C++, las herramientas comerciales existentes incluyen la prueba de mutaciones dentro de un conjunto de técnicas de prueba. Es decir, no se centran en la prueba de mutaciones y además no cubren las mutaciones a nivel de clase sino solo algunas operaciones de carácter estándar, utilizando la técnica en de manera selectiva. Las herramientas que podemos encontrar son:

- **Insure++** (1998) [18]. Esta herramienta utiliza la prueba de mutaciones como una técnica más para mejorar la calidad del software, enfocándose especialmente en problemas de memoria, pero solo realiza algunas mutaciones estándar [19]. Su enfoque es diferente al de la prueba clásica de mutaciones porque solo crea mutantes funcionalmente equivalentes, que se espera que pasen las pruebas en lugar de fallarlas. Por lo tanto, se detecta un error en el programa original cuando se mata un mutante, revelando las ambigüedades que podrían existir. Los usuarios pueden elegir la cantidad de mutantes a generar y aplicar la prueba de mutaciones a una sola función o a un proyecto completo.

- **PlexTest** (2005) [20]. Tal y como se menciona en su página web, este software implementa una prueba de mutaciones altamente selectiva para evitar la generación de mutantes equivalentes. Al seleccionar esta opción, la herramienta solo realiza la mutación de eliminación, eliminando sentencias y subexpresiones. Esta herramienta incorpora algunas otras características para mejorar el rendimiento, como la combinación con un sistema de control de revisión para determinar el código editado recientemente y probar ese código de forma selectiva.

- **Certitude Functional Qualification System** (2006) [21]. Esta herramienta combina prueba de mutaciones y análisis estático, calificando un programa funcionalmente y revelando fallos que de otro modo no podrían detectarse. Aunque este producto también se ha utilizado para el análisis de sistemas de software, ahora se dirige a la industria de la microelectrónica.

En cuanto a los sistemas de código abierto, CCMutator [22] es un generador de mutaciones para construcciones de concurrencia en C o C ++ recientemente desarrollado. Esta herramienta implementa un conjunto de operadores específicamente diseñados para mutar aplicaciones de subprocesos múltiples.

En cambio, para otros lenguajes que no sean C++, pueden encontrarse múltiples herramientas de software libre, entre las que encontramos aquellas destinadas al lenguaje de programación Java:

- **MuJava** (2013) [23] es muy utilizado en este proyecto dado su implicación en documentos de carácter científico, en los que se expresan los diferentes mutantes que realizan cada uno de sus operadores. Es un sistema de mutación para programas

Java. Genera automáticamente mutantes tanto para las pruebas de mutación tradicionales como para las pruebas de mutación a nivel de clase. MuJava puede probar clases individuales y paquetes de múltiples clases.

- **PIT** (2019) [24]. Es un sistema de prueba de mutación de última generación que proporciona una gran cantidad de operadores de mutación para Java y la JVM. Es rápido, escalable y se integra con herramientas modernas de prueba y creación.

- **Major** (2018) [25]. permite un análisis de mutación eficiente de grandes sistemas de software, así como una investigación fundamental sobre las pruebas de mutación. Major proporciona una integración en el compilador para una inserción de fallos más sencilla y rápida, además de un lenguaje específico de dominio para configurar el proceso de mutación y un analizador de mutaciones para pruebas JUnit.

- **Jester** (2005) [26]. Jester encuentra código que no está cubierto por las pruebas realizadas por el desarrollador, realiza algún cambio en su código, ejecuta sus pruebas y, si las pruebas pasan, el programa muestra un mensaje indicando el cambio.

- **Javalanche** (2012) [27]. Javalanche fue desarrollado por un equipo con sede en la Universidad de Sarland en Alemania. Javalanche está construido para la eficiencia desde cero, manipulando el código de bytes directamente y permitiendo la prueba de mutación de programas de gran magnitud, abordando el problema de los mutantes equivalentes mediante la evaluación del impacto de las mutaciones en los invariantes dinámicos: cuantos más invariantes se vean afectados por una mutación, más probable es que sea útil para mejorar los conjuntos de pruebas.

Dado todo lo anterior, podemos determinar que no existen herramientas de prueba de mutaciones de libre uso que se conozcan, exceptuando la herramienta que se actualiza en el presente proyecto, MuCPP, por lo que la hace de gran interés para su uso en la investigación de este tipo de pruebas. Destacar también la gran cantidad de herramientas para otros lenguajes con poco uso en la industria, como Java, de la que se han desarrollado cinco herramientas, aunque existen otras muchas como Testooj (2007) [28] o Jumble (2001) [29], entre otras.

Capítulo 4

Los operadores de mutación

En el presente capítulo se explicarán las diferentes clases de operadores de mutación que existen, además de los operadores que se pueden encontrar en diferentes herramientas como son MuCPP, MuJava o Major. Destacar que se escogen las herramientas de Java MuJava y Major como referencia en este trabajo ya que, al ser Java un lenguaje muy utilizado, ha sido centro de investigación de gran parte de los trabajos de prueba de mutaciones, por lo que en torno a este lenguaje se pueden encontrar herramientas más maduras que para otros lenguajes. Además, no se encuentran tantas dificultades dada la relativa similitud de Java y C++ como lenguajes orientados a objetos. La comparativa entre los operadores de las diferentes herramientas se realizará en el siguiente capítulo.

4.1. Tipos de operadores de mutación

Los operadores de mutación son reglas bien definidas sobre estructuras sintácticas para realizar determinados cambios en el código fuente de un software. Estos operadores pueden ser de diversos tipos, entre los que se encuentran los operadores tradicionales y los operadores de clase, este último contendría diversos subtipos.

- **Operadores de clase**. Operadores que realizan sus acciones sobre clases. Estos operadores pueden ser subdivididos en diferentes tipos, los cuales, son explicados a continuación.

  De herencia. Se centran en todo lo relacionado con la herencia, desde cambiar la llamada a métodos entre clases herederas a eliminar métodos de clases herederas.

  De polimorfismo y enlace dinámico. En esta clase se centran en la polimorfia, y todo lo relacionado con esta.

  De sobrecarga. Se centran en la sobrecarga de métodos.

  De reemplazo. Donde se realizan realizan reemplazos en miembros y objetos de las diferentes clases.

  De manejo de excepciones. Esta clase de operadores se centran en las excepciones, haciendo modificaciones en los manejadores de estas.

  Miscelánea. En esta clase de operadores se asignan todos aquellos que no se han asignado en un tipo anterior.

- **Tradicionales**. Operadores que realizan sus acciones sobre métodos concretos o líneas de código fuente concretas. Son denominados como tradicionales ya que fueron los operadores de mutación creados en los comienzos de la técnica y porque abordan elementos que suelen ser comunes a la mayor parte de lenguajes de programación.

4.2. Los operadores de mutación en MuCPP

La herramienta de prueba de mutaciones denominada MuCPP es creada en el seno de la Universidad de Cádiz, concretamente en su Grupo de Investigación UCASE de Ingeniería del Software [1]. Esta herramienta, en un principio, generaba solamente mutantes tradicionales, pasando posteriormente a generar mutantes a nivel de clases mediante diferentes mejoras, además de las mejoras otorgadas para su fácil manejo en los diferentes años de su existencia.

MuCPP tiene un sencillo funcionamiento, el cual es explicado en el Apéndice A de este documento, además de contar con una gran cantidad de operadores que le hacen ser un gran competidor frente a otras herramientas para el mismo u otros lenguajes de programación.

En la herramienta de prueba de mutaciones MuCPP encontramos todos los tipos de operadores expresados en el apartado anterior siendo, tal y como expresa P. Delgado et al. en su artículo [4], los que podemos encontrar a continuación.

4.2.1. Operadores de herencia

Bloque	Operador	Descripción
De herencia	IHD	Borrado de variable oculta
	IHI	Inserción de variable oculta
	ISD	Borrado de referencia a clase padre
	ISI	Inserción de referencia a clase padre
	IOD	Borrado de método sobrescrito
	IOP	Cambio de posición de la llamada a método sobrescrito
	IOR	Renombrado de método sobrescrito
	IPC	Borrado de llamada explícita al constructor de la clase padre
	IMR	Reemplazo de herencia múltiple

Tabla 2: Tabla operadores de herencia en MuCPP.

Tal y como podemos observar en la tabla 2 existen nueve operadores relacionados con la herencia en MuCPP, entre los que se encuentran:

[IHD] Borrado de variable oculta

Este operador, tal y como indica su nombre, elimina la definición de una variable oculta o sobrescrita por un método hijo. De este modo las referencias a la variable se realizarán a la clase padre.

[IHI] Inserción de variable oculta

Este operador inserta la definición de una variable oculta o sobrescrita de una clase padre en una clase hijo. De este modo las referencias a la variable se realizarán a la clase hijo y no a la clase padre.

[ISD] Borrado de referencia a clase padre

Mediante este operador, tal y como se indica en su nombre, se elimina la referencia a la clase padre.

[ISI] Inserción de referencia a clase padre

Mediante este operador se inserta la referencia a la clase padre desde una clase hijo cuando se llama a un atributo de la clase.

[IOD] Borrado de método sobrescrito

Mediante este operador se simula un fallo del olvido de sobrescribir un método de una clase padre en una clase hija.

[IOP] Cambio de posición de la llamada al método sobrescrito

Este operador cambia la posición de la llamada a un método padre desde un método de la clase hijo.

[IOR] Renombrado método sobrescrito

El cometido de este operador es renombrar el método de la clase padre, de forma, que el método sobrescrito en la clase hijo no afecte a la clase padre de ninguna forma.

[IPC] Borrado de llamada explícita al constructor de la clase padre

Este operador, tal y como indica su nombre, elimina la llamada explícita por parte de una clase hijo al constructor de una clase padre.

[IMR] Reemplazo de herencia múltiple

Este operador tiene como cometido la modificación o renombrado de la clase a la que se accede para ejecutar un método. Esto sucede cuando una clase hereda de dos o más clases y estas tienen un método en común al que se desea acceder.

4.2.2. Operadores de polimorfismo y enlace dinámico

Bloque	Operador	Descripción
Operadores de polimorfismo y enlace dinámico	PVI	Inserción del modificador virtual
	PCD	Eliminación del operador de modelado de tipos
	PCI	Inserción del operador de modelado de tipos
	PCC	Cambio del tipo de modelado
	PMD	Declaración con tipo de clase padre
	PPD	Declaración de una variable parámetro con el tipo de una clase hija
	PNC	Llamada al método new con tipo de clase hija
	PRV	Asignación de una referencia con otro tipo compatible

Tabla 3: Tabla de operadores de polimorfismo y enlace dinámico en MuCPP.

Tal y como podemos observar en la tabla 3 existen ocho operadores relacionados con el polimorfismo y el enlace dinámico en MuCPP, entre los que se encuentran:

[PVI] Inserción del modificador virtual

Mediante este operador se asegura la posibilidad de que el programador olvide incluir en la declaración de un método la palabra *virtual*, la cual se necesita para emplear el polimorfismo en C++.

[PCD] Eliminación del operador de modelado de tipos

Este operador realiza la acción contraria a la realizada por la inserción del operador PCI, el cual inserta este mismo operador de modelado.

[PCI] Inserción del operador de modelado de tipos

Es el contrario al operador anterior, donde se inserta el modelado de tipos. Esta acción se realiza mediante el operador *dynamic_cast*.

[PCC] Cambio del tipo de modelado

Mediante este operador no se realiza un modelado ni se elimina uno, sino se modifica el modelado, cambiando la clase a la que se convertirá por otra que contenga un método con el mismo nombre.

[PMD] Declaración con tipo de clase padre

Mediante este operador se cambiará la declaración de un tipo hijo a un tipo padre.

[PPD] Declaración de una variable parámetro con el tipo de una clase hija

Este operador realiza la misma acción que el operador anterior, pero en lugar de declarar la variable como miembro, esta se declara en la llamada a un método o función, a la que se le pasa como parámetro.

[PNC] Llamada al método *new* con tipo de clase hija

Este operador, cambia la llamada al método constructor de la clase, llamando al constructor de la clase hijo. Esto simula un posible fallo de programación en el que el programador se ha confundido al realizar la llamada y llama a la clase padre en lugar de a la clase hijo.

[PRV] Asignación de una referencia con otro tipo compatible

Este operador modifica la asignación de una referencia mediante otra con un tipo compatible. Es habitual el error en la asignación de un hijo a una clase padre y equivocarse de hijo, es por ello, que existe este operador, el cual intentará encontrar al hijo correcto creando diversos mutantes entre todos los hijos declarados y compatibles.

4.2.3. Operadores de sobrecarga

Bloque	Operador	Descripción
De sobrecarga de métodos	OMD	Borrado de método sobrecargado
	OMR	Cambio del contenido del método sobrecargado
	OAN	Cambio del número de argumentos
	OAO	Cambio del orden de los argumentos

Tabla 4: Tabla operadores de sobrecarga en MuCPP.

Tal y como podemos observar en la tabla 4 existen cuatro operadores relacionados con la sobrecarga en MuCPP.

[OMD] Borrado del método sobrecargado

Este operador crea mutantes eliminando uno de los métodos sobrecargado con los que cuente la clase. Si el mutante aún sigue vivo tras su ejecución significará que el método al que llama es el incorrecto y por consiguiente se está produciendo una conversión del tipo de parámetro incorrecta.

[OMR] Cambio del contenido del método sobrecargado

Este operador tiene como fin el comprobar que se está realizando la llamada al método correcto, por ello se reemplaza el contenido de un método por el contenido de otro con el mismo nombre.

[OAN] Cambio del número de argumentos

Otra manera de asegurar que el programador no invoca el método incorrecto debido a la sobrecarga de métodos es cambiar el número de argumentos en la invocación al método. Análogamente al caso anterior, debe existir la sobrecarga que acepte el cambio en la lista de argumentos.

[OAO] Cambio del orden de los argumentos

Este operador modifica el orden de los argumentos, comprobando así que no exista otro método capaz de ejecutar esa llamada, lo cual sería un error de programación.

4.2.4. Operadores de manejo de excepciones

Bloque	Operador	Descripción
Manejo de excepciones	EHC	Cambio del manejador de la excepción
	EHR	Eliminación del manejador de excepción

Tabla 5: Tabla operadores del manejo de excepciones en MuCPP.

Tal y como podemos observar en la tabla 6 existen dos operadores relacionados con las excepciones en MuCPP, entre los que se encuentran:

[EHC] Cambio del manejador de la excepción

En MuCPP este operador crea una excepción en lugar de ser tratada en ese momento, modificando así ese tratamiento y propagando la excepción, para que, con suerte, sea tratada en otro lugar de la ejecución del programa.

[EHR] Eliminación del manejador de excepción

El operador EHR modifica el manejador de la excepción eliminando una de las cláusulas catch, siempre que haya más de una. Al eliminar el manejador, se pospone la captura de la excepción pasándola al manejador más cercano. De esta manera, se comprueba si el programador realiza correctamente el manejo de la excepción.

4.2.5. Operadores de reemplazo

Bloque	Operador	Descripción
De reemplazo	MCO	Llamada a miembro de otro objeto
	MCI	Llamada a miembro de la clase que se hereda

Tabla 6: Tabla operadores del reemplazo en MuCPP.

Tal y como podemos observar en la tabla 6 existen dos operadores relacionados con las excepciones en MuCPP, entre los que se encuentran:

[MCO] Llamada a miembro de otro objeto

El concepto de este operador es muy simple y se basa en el erro de uso de un objeto, en este caso se modifica el objeto al que se realiza la llamada a un método, modificando así a otro objeto de la misma clase.

[MCI] Llamada a miembro de la clase que se hereda

Este operador es un poco más complicado, dado que se llama a métodos de tipo *virtual*, es decir, métodos de la clase de la que se hereda y se llamará con otro objeto de otra clase pero que heredan de la misma.

4.2.6. Operadores de miscelánea

Bloque	Operador	Descripción
Miscelánea	CTD	Eliminación de la palabra clave this
	CTI	Inserción de la palabra clave this
	CID	Eliminación de la inicialización de una variable
	CDC	Creación del constructor por defecto
	CDD	Eliminación del destructor de clase
	CCA	Eliminación del constructor de copia y del constructor de asignación

Tabla 7: Tabla operadores de miscelánea en MuCPP.

Tal y como podemos observar en la tabla 7 existen seis operadores de esta clase en MuCPP, entre los que se encuentran:

[CTD] Eliminación de la palabra clave this

El operador de eliminación de la palabra clave *this* elimina las ocurrencias que existan en el código fuente sobre esta palabra clave.

[CTI] Inserción de la palabra clave this

Este operador se considera de gran importancia por el cambio del entorno de las expresiones utilizadas en los diferentes códigos fuente.

[CID] Eliminación de la inicialización de una variable

Mediante este operador se elimina la inicialización de una variable dada, este es otro de los errores comunes de programación y que se suelen observar habitualmente, dado que el propio compilador asigna una dirección de memoria a la variable, el cual ha podido estar ya asignado a otra variable anteriormente y contener "basura".

[CDD] Eliminación del destructor de clase

Otro de los fallos de programación habituales son el olvidar la construcción del destructor, dado que en otros lenguajes de programación como Java el recolector de basura realiza este trabajo por nosotros, este olvido es muy habitual. Es por ello que se crea este

mutante simulando este error tan común en la programación.

[CDC] Creación del constructor por defecto

Es un error muy común no establecer un constructor, dado que existe el constructor por defecto, pero esto suele ser una mala práctica, es por ello que se crea este mutante en caso de que el constructor se haya creado sin ningún parámetro, dejando así el constructor por defecto y con ello un error de programación muy frecuente.

[CCA] Eliminación del constructor de copia y del constructor de asignación

Es muy común en el lenguaje de programación C++ la creación del constructor de copia y el constructor de asignación, dado que los constructores por defecto no realizan el trabajo de copia como tal, sino que asignan la dirección de memoria del otro objeto. Es por ello que se crea este mutante, simulando el olvido de la creación del constructor de copia y asignación, haciendo que con cada uso de la copia y asignación se asigne la dirección de memoria y no se copie cada uno de los elementos internos del objeto.

4.2.7. Operadores tradicionales

Bloque	Operador	Descripción
Tradicionales	ARB	Operador de reemplazo aritmético (+, -, *, /, %)
	ARU	Operador de reemplazo de operadores unarios (+, -)
	ARS	Operador de reemplazo de operaciones aritméticas con atajos (++, −−)
	AIU	Operador de inserción de operadores unarios (-)
	AIS	Operador de inserción de atajos (++, −−)
	ADS	Operador de eliminación de atajos (++, −−)
	ROR	Operador de reemplazo de operadores relacionales(<, <=, >, >=, ==, !=)
	COR	Operador de reemplazo de operadores condicionales (&&, ||)
	COD	Operador de eliminación de operadores condicionales (!)
	COI	Operador de inserción de operadores condicionales (!)
	LOR	Operador de reemplazo de operadores lógicos (|, ^)
	ASR	Operador de reemplazo atajos con asignación (%)

Tabla 8: Tabla operadores tradicionales en MuCPP.

Tal y como podemos observar en la tabla 8 existen doce operadores tradicionales en MuCPP, entre los que se encuentran:

[AOR] Operador de reemplazo aritmético

El operador operador de Reemplazos aritméticos realiza cambios entre los diferentes operadores aritméticos conocidos.

[ARU] Operador de reemplazo de operadores unarios

Este tipo de operaciones de reemplazo son muy similares a las expresadas en el apartado anterior, siendo para el operador unario + modificado por - y viceversa.

[ARS] Operador de reemplazo de operaciones aritméticas con atajos

Este operador, al igual que los anteriores, modifica los operadores unarios $++$ y $--$ puestos tanto delante del operadorando al que se realiza la operación como detrás.

[AIU] Operador de inserción de operadores unarios

MuCPP crea un mutante en el caso de encontrar un operando sin nada más, insertando delante de este el operador unario $-$.

[AIS] Operador de inserción de atajos

Este operador inserta los operadores unarios $++$ y $--$ tanto delante como detrás de los operandos.

[ADS] Operador de eliminación de atajos

Este operador elimina los operadores unarios $++$ y $--$ que se encuentran tanto delante como detrás de los operandos.

[ROR] Operador de reemplazo de operadores relacionales

El operador ROR u operador de reemplazo de operadores relacionales realiza cambios entre los diferentes operadores relacionales más básicos.

[COR] Operador de reemplazo de operadores condicionales

El operador COR u operador de reemplazo de operadores condicionales realiza cambios entre los diferentes operadores condicionales más básicos.

[COD] Operador de eliminación de operadores condicionales

El operador COD u operador de eliminación de operadores condicionales, elimina como su nombre indica la negación de un operando.

[COI] Operador de inserción de operadores condicionales

El operador COI u operador de inserción de operadores condicionales, inserta como su nombre indica la negación de un operando.

[LOR] Operador de reemplazo de operadores lógicos

El operador LOR u operador de reemplazo de operadores lógicos, modifica como su nombre indica los diferentes operadores lógicos que podemos encontrar en C++.

[ASR] Operador de reemplazo atajos con asignación

El operador ASR u operador de reemplazo atajos con asignación, modifica como su nombre indica los diferentes operadores con asignación que podemos encontrar en C++.

4.3. Los operadores de mutación en MuJava

Los operadores en la herramienta MuJava se encuentran divididos en dos grupos bien definidos, siendo estos los operadores relacionados con clases y por otro lado los operadores relacionados con funciones o métodos. Es por ello que se han dividido los operadores en estos dos grupos, los cuales podemos encontrar en las secciones siguientes. Estos operadores han sido obtenidos de los artículos de investigación [30] y [31]. Destacar que estos operadores no son explicados como en el caso de MuCPP ya que muchos de ellos ya han sido definidos anteriormente o son de fácil entendimiento con su descripción.

4.3.1. Operadores sobre clases

Estos operadores son aquellos que se realizan sobre clases y tienen diferentes subtipos bien definidos y los cuales podemos ver en las tablas siguientes.

Clase	Operador	Descripción
Encapsulación	AMC	Operador de modificación de acceso

Tabla 9: Tabla operadores de encapsulación en muJava.

Tal y como podemos observar en la tabla 9 existe un solo operador de encapsulación en MuJava.

Clase	Operador	Descripción
Herencia	IHI	Operador de inserción de variable sobrescrita
	IHD	Operador de eliminación de variable sobrescrita
	IOD	Operador de eliminación de método sobrescrito
	IOP	Operador de cambio de posición de llamada a método sobrescrito
	IOR	Operador de renombrado de método sobrescrito
	ISI	Operador de inserción de palabra clave super
	ISD	Operador de eliminación de palabra clave super
	IPC	Operador de eliminación de llamada explícita al constructor de la clase padre

Tabla 10: Tabla operadores de herencia en muJava.

Existen un total de ocho operadores de herencia en MuJava, tal y como podemos comporbar en la tabla 10.

Clase	Operador	Descripción
Polimorfismo	PNC	Operador de llamada al método new con tipo de la clase hijo
	PMD	Operador de declaración de variable con tipo de la clase padre
	PPD	Operador de declaración de parámetro con tipo de la clase hijo
	PCI	Operador de inserción del operador de modelado de tipos
	PCD	Operador de eliminación del operador de modelado de tipos
	PCC	Operador de cambio del tipo del modelado de tipos
	PRV	Operador de cambio de referencia con un tipo compatible
	OMR	Operador de reemplazo de contenido de método sobrescrito
	OMD	Operador de eliminación de método sobrescrito
	OAC	Cambio de argumentos del método sobrecargado

Tabla 11: Tabla operadores de polimorfismo en muJava.

Tal y como podemos observar en la tabla 11 existen diez operadores de polimorfismo en MuJava.

Clase	Operador	Descripción
Características específicas de Java	JTI	Operador de inserción de la palabra clave this
	JTD	Operador de eliminación de la palabra clave this
	JSI	Operador de inserción del modificador static
	JSD	Operador de eliminación del modificador static
	JID	Operador de eliminación de inicialización de variable
	JDC	Operador de creación del constructor por defecto
	EOA	Operador de asignación de referencia y reemplazo de asignación de contenido
	EOC	Operador de comparación de referencia y reemplazo de comparación de contenido
	EAM	Operador de cambio del método de acceso
	EMM	Operador de cambio de método modificado

Tabla 12: Tabla operadores de características específicas de Java en muJava.

Existen diez características específicas de java entre los operadores de MuJava, tal y como podemos ver en la tabla 12.

4.3.2. Operadores sobre funciones o métodos

Operador	Descripción
AOR	Operador de reemplazo de operadores aritméticos
AOI	Operador de inserción de operadores aritméticos
AOD	Operador de eliminación de operadores aritméticos
ROR	Operador de reemplazo de operadores relacionarles
COR	Operador de reemplazo de operadores condicionales
COI	Operador de inserción de operadores condicionales
COD	Operador de eliminación de operadores condicionales
SOR	Operador de reemplazo de operadores de desplazamiento
LOR	Operador de reemplazo de operadores lógicos
LOI	Operador de inserción de operadores lógicos
LOD	Operador de eliminación de operadores lógicos
ASR	Operador de reemplazo de operadores de asignación
SDL	Operador de eliminación de sentencias
VDL	Operador de eliminación de variables
CDL	Operador de eliminación de constantes
ODL	Operador de eliminación de operadores

Tabla 13: Tabla operadores sobre métodos en muJava.

En esta sección encontramos todos y cada uno de los operadores expresados en la tabla 13, los cuales son realizados sobre métodos o funciones específicas, lo que nos lleva a denominarlos como operadores tradicionales.

4.4. Los operadores de mutación en Major

Los operadores que podemos encontrar en la herramienta Major se basan todos ellos en operadores tradicionales, dejando de lado los operadores de otro tipo. Por ello, a continuación se presentan todos los operadores en la tabla 14, que han sido obtenidos del articulo [32]. Destacar que estos operadores no son explicados como en el caso de MuCPP ya que muchos de ellos ya han sido definidos anteriormente o son de facil entendimiento con su descripción.

Operador	Descripción
AOR	Operador de reemplazo de operadores aritméticos
ROR	Operador de reemplazo de operadores relacionarles
COR	Operador de reemplazo de operadores condicionales
SOR	Operador de reemplazo de operadores de desplazamiento
LOR	Operador de reemplazo de operadores lógicos
SDL	Operador de eliminación de sentencias
LVR	Operador de reemplazo de valores literales
ORU	Operador de reemplazos unanios

Tabla 14: Tabla operadores Major.

4.5. Tabla operadores de mutación en las herramientas MuCPP, MuJava y Major

Operador	Aparece en MuCPP	Aparece en MuJava	Aparece en Major
IHD	X	X	
IHI	X	X	
ISD	X	X	
ISI	X	X	
IOD	X	X	
IOP	X	X	
IOR	X	X	
IPC	X	X	
IMR	X		
PVI	X		
PCD	X	X	
PCI	X	X	
PCC	X	X	
PMD	X	X	
PPD	X	X	
PNC	X		
PRV	X	X	
OMR	X	X	
OMD	X	X	
OAN	X	X	
OAO	X	X	
OAC	X	X	
EHC	X		
EHR	X		
MCO	X		
MCI	X		
CTD	X	X	
CTI	X	X	
CID	X	X	
CDC	X	X	
CDD	X		
CCA	X		
ARB	X	X	X
ARU	X	X	X
ARS	X	X	X
AIU	X	X	X
AIS	X	X	X
ADS	X	X	X
ROR	X	X	X
COR	X	X	X
COD	X	X	X
COI	X	X	X
LOR	X	X	X
ASR	X	X	X
JSI		X	
JSD		X	
EOA		X	
EOC		X	
EAM		X	
EMM		X	
SOR		X	X
SDL		X	X
VDL		X	
CDL		X	
ODL		X	
LVR			X
AMC		X	

Tabla 15: Tabla comparativa operadores.

En la tabla 15 podemos observar una comparativa entre las tres herramientas que se han utilizado para el análisis en el siguiente capítulo, marcando con una **X** en el caso en el que se encuentre en la herramienta y en blanco en caso de que no se encuentre en ella.

Destacar que en la tabla 15 no se han incluido algunos operadores dado que en la herramienta MuCPP estos operadores poseen otro nombre, los cuales han sido indicados en la tabla 16.

Operador en MuCPP	**Operador en MuJava**
CTD	JTD
CTI	JTI
CID	JID
CDC	JDC
ARB	AOR
ARU	ORU
AIU + AIS	AOI
ADS	AOD

Tabla 16: Tabla de cambio de nombres de operadores entre MuCPP y MuJava.

Capítulo 5

Comparación y actualización de los operadores de MuCPP con los de otras herramientas

Dado el amplio abanico de posibilidades para los diferentes operadores de mutación que podemos encontrar en la actualidad, este proyecto compara los operadores de diferentes herramientas con los operadores de MuCPP, detectando las diferencias encontradas y determinando si estas son importantes para el fin de la herramienta o no. Para ello se ha contemplado múltiples investigaciones como son [33] [34] [35] [36] [30] [37] [38] [31], haciendo uso de esos artículos.

Destacar que algunos operadores encontrados en las otras herramientas no aparecen en MuCPP y no han sido contemplados en esta comparativa. Estos operadores han sido analizados e implementados en su mayoría y serán explicados en el capítulo 6.

Para la realización de esta comparativa se han observado diversas herramientas de prueba de mutaciones, como son muJava, no actualizada desde junio de 2013, pero con una gran cantidad de operadores en comparación con la competencia y Major, actualizado el pasado año (2018) pero con una cantidad de operadores menor. En las herramientas muJava y Major encontramos diversos operadores, los cuales se comparan uno a uno con los que se contemplan en MuCPP, empezando por los más básicos y terminando por los más complejos. En concreto, se comparan los siguientes tipos de operadores:

- **Operadores tradicionales**. Estos se recogen en la tabla 17 indicando en cada uno de ellos si ha sido actualizado en MuCPP.
- **Operadores sobre clases**. Estos se recogen en la tabla 18 los cuales se categorizan en los siguientes tipos y no han sido necesario actualizarlos:

  Operadores de herencia.

  Operadores de polimorfismo y enlace dinámico.

  Operadores de sobrecarga.

  Operadores de manejo de excepciones.

  Operadores de reemplazo.

Operadores de miscelánea.

Operador	Actualizado
[AOR] Operador de reemplazo aritmético	SÍ
[ARU] Operador de reemplazo de operadores unarios	SÍ
[ARS] Operador de reemplazo de operaciones aritméticas con atajos	SÍ
[AIU] Operador de inserción de operadores unarios	SÍ
[AIS] Operador de inserción de atajos	NO
[ADS] Operador de eliminación de atajos	NO
[ROR] Operador de reemplazo de operadores relacionales	SÍ
[COR] Operador de reemplazo de operadores condicionales	SÍ
[COD] Operador de eliminación de operadores condicionales	NO
[COI] Operador de inserción de operadores condicionales	NO
[LOR] Operador de reemplazo de operadores lógicos	SÍ
[ASR] Operador de reemplazo atajos con asignación	SÍ

Tabla 17: Tabla operadores tradicionales comparados.

Operador	Subtipo
[IHD] Borrado de variable oculta	De herencia
[IHI] Inserción de variable oculta	De herencia
[ISD] Borrado de referencia a clase padre	De herencia
[ISI] Inserción de referencia a clase padre	De herencia
[IOD] Borrado de método sobrescrito	De herencia
[IOP] Cambio de posición de la llamada al método sobrescrito	De herencia
[IOR] Renombrado método sobrescrito	De herencia
[COR] Operador de reemplazo de operadores condicionales	De herencia
[IPC] Borrado de llamada explícita al constructor de la clase padre	De herencia
[IMR] Reemplazo de herencia múltiple	De herencia
[PVI] Inserción del modificador virtual	De polimorfismo y enlace dinámico
[PCD] Eliminación del operador de modelado de tipos	De polimorfismo y enlace dinámico
[PCI] Inserción del operador de modelado de tipos	De polimorfismo y enlace dinámico
[PCC] Cambio del tipo de modelado	De polimorfismo y enlace dinámico
[PMD] Declaración con tipo de clase padre	De polimorfismo y enlace dinámico
[PPD] Declaración de una variable parámetro con el tipo de una clase hija	De polimorfismo y enlace dinámico
[PNC] Llamada al método new con tipo de clase hija	De polimorfismo y enlace dinámico
[PRV] Asignación de una referencia con otro tipo compatible	De polimorfismo y enlace dinámico
[OMD] Borrado del método sobrecargado	De sobrecarga
[OMR] Cambio del contenido del método sobrecargado	De sobrecarga
[OAN] Cambio del número de argumentos	De sobrecarga
[OAO] Cambio del orden de los argumentos	De sobrecarga
[EHC] Cambio del manejador de la excepción	De manejo de excepciones
[EHR] Eliminación del manejador de excepción	De manejo de excepciones
[MCO] Llamada a miembro de otro objeto	De reemplazo
[MCI] Llamada a miembro de la clase que se hereda	De reemplazo
[CTD] Eliminación de la palabra clave this	De miscelánea
[CTI] Inserción de la palabra clave this	De miscelánea
[CID] Eliminación de la inicialización de una variable	De miscelánea
[CDD] Eliminación del destructor de clase	De miscelánea
[CDC] Creación del constructor por defecto	De miscelánea
[CCA] Eliminación del constructor de copia y del constructor de asignación	De miscelánea

Tabla 18: Tabla operadores comparados por subtipos.

5.1. Comparación de operadores tradicionales

En primer lugar comenzaremos por los operadores tradicionales, los cuales podemos encontrar en su mayoría en las tres herramientas. Estos operadores son actualizados tras cada comparativa dada la importancia que tienen.

5.1.1. [AOR] Operador de reemplazo aritmético

El operador **AOR** u operador de reemplazos aritméticos, realiza cambios entre los diferentes operadores aritméticos conocidos. En la tabla 19 se pueden encontrar los diferentes mutantes generados en cada una de las herramientas, siendo op1 y op2 cada uno de los operandos de la operación binaria correspondiente.

Código Inicial	**Mutante en MuCPP**	**Mutantes en MuJava**						**Mutantes en Major**			
op1 + op2	op1 - op2	op1 - op2	op1 * op2	op1 / op2	op1 % op2	op1	op2	op1 - op2	op1 * op2	op1 / op2	op1 % op2
op1 - op2	op1 + op2	op1 + op2	op1 * op2	op1 / op2	op1 % op2	op1	op2	op1 + op2	op1 * op2	op1 / op2	op1 % op2
op1 * op2	op1 / op2	op1 + op2	op1 - op2	op1 / op2	op1 % op2	op1	op2	op1 + op2	op1 - op2	op1 / op2	op1 % op2
op1 / op2	op1 * op2	op1 + op2	op1 - op2	op1 * op2	op1 % op2	op1	op2	op1 + op2	op1 - op2	op1 * op2	op1 % op2
op1 % op2	op1 / op2	op1 + op2	op1 - op2	op1 * op2	op1 / op2	op1	op2	op1 + op2	op1 - op2	op1 * op2	op1 / op2

Tabla 19: Tabla comparativa operador AOR con otras herramientas.

La herramienta en la que se está trabajando en este TFG, MuCPP, se encuentra más desactualizada que el resto de herramientas ya que genera un único mutante en comparación con MuJava, la cual genera seis mutantes con cada uso y Major que genera cuatro mutantes con cada uso. Este operador realizaba hasta ahora los mismos mutantes que el operador correspondiente de la herramienta PIT, pero en la actualidad esta herramienta también crea todos los mutantes que podemos ver en MuJava o Major.

En primer lugar la herramienta MuCPP crea el mutante opuesto al que se encontraba en el código inicial, es decir, si encontramos inicialmente un *+*, creará un mutante con un - y viceversa, lo mismo pasa con la operación * modificada por un / y viceversa o la operación % modificada por un /.

Las otras herramientas usan todos y cada uno de los operadores clásicos *+, -, *, / y %* y crean los cuatro mutantes con todos los operadores quitando el que estaba puesto en un inicio. En este punto habría que destacar que la herramienta MuJava también genera dos mutantes diferentes quitando en cada uno de los casos el operador y uno de los dos operandos.

Tras la comparativa se considera necesaria la actualización de este operador en MuCPP, que pasaría a crear todos y cada uno de los mutantes que crea la herramienta Major, ya que son todos los mutantes que existen en ambas herramientas, además de considerar que con el aumento de mutantes generados se podrán dar más casos no contemplados en las pruebas realizadas por el desarrollador.

No se decide incluir los mutantes que quita uno de los dos operandos y el operador ya que se creará en el capítulo siguiente un operador que realiza estos mutantes, denominado ODL. Tras lo explicado, se considera que la tabla 20 resume de manera correcta las acciones que realiza el operador tras la actualización.

Código Inicial	**Mutantes en MuCPP actualizado**			
op1 + op2	op1 − op2	op1 * op2	op1 / op2	op1 % op2
op1 − op2	op1 + op2	op1 * op2	op1 / op2	op1 % op2
op1 * op2	op1 + op2	op1 − op2	op1 / op2	op1 % op2
op1 / op2	op1 + op2	op1 − op2	op1 * op2	op1 % op2
op1 % op2	op1 + op2	op1 − op2	op1 * op2	op1 / op2

Tabla 20: Tabla operador AOR tras su actualización.

5.1.2. [ARU] Operador de reemplazo de operadores unarios

El operador **ARU** u operador de reemplazo de operadores unarios, realiza cambios entre los diferentes operadores unarios conocidos. En la tabla 21 se pueden encontrar los diferentes mutantes generados en cada una de las herramientas, siendo op el operando del operador unario correspondiente.

Código Inicial	**Mutante en MuCPP**	**Mutantes en MuJava**					**Mutantes en Major**	
+op	−op							
−op	+op	op	−−op	op−−	op++	++op	op	0

Tabla 21: Tabla comparativa operador ARU con otras herramientas.

Este tipo de operaciones de reemplazo son muy similares a las expresadas en el apartado anterior, contemplando un gran número de mutaciones en MuJava y un número menor en Major, pero MuCPP sigue siendo el que menor número de expresiones contempla, siendo para el operador unario + modificado por - y viceversa. En cambio, en MuJava cambia los operadores unarios por los atajos unarios y en Major lo cambia por el cero, en el caso del $-op$, además de quitar el propio operador y no poner ninguno.

Tras observar el operador ARU en las diferentes herramientas, se decide que lo correcto es la actualización en MuCPP del operador para que realice todas las mutaciones contempladas en MuJava. Destacar que otras herramientas no contemplan la opción de que un operador aparezca como *+op* y realizarle los mutantes correspondientes, es por ello que se decide contemplar los mutantes del operador en el caso de *-op* para ambos casos.

Con esta actualización queda este operador completamente actualizado incluyendo todas las mutaciones que poseen sus principales competidores y contemplando casos que no contemplan otras herramientas, tal y como se muestra en la tabla 22. Con esta mejora se multiplica por cinco el numero de mutaciones creado cada vez que se usa este operador en la herramienta MuCPP.

Código Inicial	**Mutantes en MuCPP actualizado**				
+op	−op	−−op	op−−	op++	++op
−op	+op	−−op	op−−	op++	++op

Tabla 22: Tabla operador ARU tras su actualización.

5.1.3. [ARS] Operador de reemplazo de operaciones aritméticas con atajos

El operador **ARS** u operador de reemplazo de operaciones aritméticas con atajos, realiza cambios entre los diferentes operadores unarios de atajos. En la tabla 23 se pueden encontrar los diferentes mutantes generados en cada una de las herramientas, siendo op el operando del operador unario correspondiente.

Código Inicial	**Mutantes en MuCPP**		**Mutantes en MuJava**				**Mutantes en Major**
op++	op−−	++op	++op	op−−	−−op	op	
++op	−−op	op++	op−−	−−op	op++	op	
op−−	op++	−−op	++op	−−op	op++	op	
−−op	++op	op−−	++op	op−−	op++	op	

Tabla 23: Tabla comparativa operador ARS con otras herramientas.

Este operador, al igual que los anteriores no contempla todos los casos posibles en nuestra herramienta, en comparación con el principal competidor. Cabe destacar que Major no contempla este operador o conjunto de reemplazos aritméticos.

Nuestra herramienta modifica el operador unario ++ posterior al operando por el mismo pero anterior al operando y el −− en el mismo lugar. Lo mismo pasa con el −− posterior al operando, cambiandolo por ++ posterior al operando y el −− anterior al operando. Por ultimo los ++ y −− anteriores al operando se cambian entre ellos y por el mismo pero posterior al operando. Esto diferencia a nuestra herramienta con MuJava la cual crea mutantes con los otros tres operadores que no son el que venía inicialmente y por otro lado, quita el operador.

Para que MuCPP se encuentre a la última en cuanto a la creación de mutantes se decide actualizar el operador, creando un mutante más con cada uso de este operador y quedando como se puede observar en la tabla 24.

Código Inicial	**Mutantes en MuCPP Actualizado**		
op++	op−−	++op	−−op
++op	−−op	op++	op++
op−−	op++	−−op	op++
−−op	++op	op−−	op++

Tabla 24: Tabla operador ARS tras su actualización.

5.1.4. [AIU] Operador de inserción de operadores unarios

El operador **AIU** u operador de inserción de operadores unarios, inserta operadores unarios cuando un operando no posee ninguno. En la tabla 25 se pueden encontrar los diferentes mutantes generados en cada una de las herramientas, siendo op el operando del operador unario correspondiente.

Código Inicial	**Mutantes en MuCPP**	**Mutantes en MuJava**					**Mutantes en Major**	
op	-op	−op	−−op	op−−	op++	++op	−op	0

Tabla 25: Tabla comparativa operador AIU con otras herramientas.

Como se ha podido observar en la tabla, MuCPP es la herramienta que crea un menor número de mutantes con el operador, quedando por detrás de Major, que crea dos y por detrás de MuJava, el cual crea un total de cinco mutantes cada vez que se usa el operador.

Cabe destacar que MuJava no modifica el operador por cero, cosa que si hace Major, modificando completamente el operador que puede influir a la hora de la devolución de resultados en métodos o funciones.

Destacar antes de realizar las modificaciones que los mutantes $--op$, $op--$, $++op$ y $op++$ que crea MuJava se crean en MuCPP mediante el operador AIS, que será analizado posteriormente. Al observar cada una de las herramientas, se decide que MuCPP debe contener todos y cada uno de los mutantes que crean ambas herramientas, quedando así un operador completo. Por ello, este operador quedará incluyendo el mutante $-op$ y 0, y el operador AIS quedará insertando los mutantes que se generaban hasta ahora, por lo que MuCPP contendrá todos los casos expuestos en la tabla 26.

Código Inicial	**Mutantes en MuCPP Actualizado**	
op	−op	0

Tabla 26: Tabla operador AIU tras su actualización.

Con esta actualización se multiplica por dos el numero de mutantes que se generan con cada uso de este operador en MuCPP.

5.1.5. [AIS] Operador de inserción de atajos

El operador **AIS** u operador de inserción de atajos, inserta atajos unarios cuando un operando no posee ningún operador unario. En la tabla 27 se puede observar como MuCPP realiza estos cambios. En otras herramientas como MuJava las acciones de este operador las realiza el operador AIU, las cuales pueden ser observadas en la tabla 25.

Código Inicial	**Mutantes en MuCPP**			
op	−−op	op−−	op++	++op

Tabla 27: Tabla mutaciones operador AIS en MuCPP.

Dado lo explicado en el operador AIU, podemos determinar que este operador se quedará de la misma forma que hasta el momento, generando los mutantes que se expresan en la tabla y complementando al operador AIU.

5.1.6. [ADS] Operador de eliminación de atajos

El operador **ADS** u operador de eliminación de atajos, elimina el operador unario que posea un operando. En la tabla 28 se pueden encontrar los diferentes mutantes generados en MuCPP mediante este operador, siendo op el operando del operador unario correspondiente.

Código Inicial	**Mutantes en MuCPP**
++op	op
−−op	op
op++	op
op−−	op

Tabla 28: Tabla mutaciones operador ADS en MuCPP.

Este operador elimina los atajos de los operadores tal y como se contempla en la tabla. Cabe destacar que otras herramientas como MuJava eliminan estos atajos mediante el operador AOD.

5.1.7. [ROR] Operador de reemplazo de operadores relacionales

Tal y como podemos observar en la siguiente tabla, MuCPP se queda muy atrasado con respecto a este operador, dado que genera un solo mutante en cada uso, en cambio, sus principales competidores generan muchos más, en el caso de Major, genera tres en algunos casos y cinco en otros, y en el caso de MuJava genera, nada más y nada menos, que nueve mutantes en cada uso de este operador, dejando a MuCPP sin posibilidades en cuanto a este operador.

Código Inicial	**Mutantes en MuCPP**	**Mutantes en MuJava**									**Mutantes en Major**				
op1>op2	op1>=op2	op1>=op2	op1<op2	op1<=op2	op1==op2	op1!=op2	T	F	op1	op2	op1>=op2	op1!=op2	F		
op1>=op2	op1>op2	op1>op2	op1<op2	op1<=op2	op1==op2	op1!=op2	T	F	op1	op2	op1>op2	op1==op2	T		
op1<op2	op1<=op2	op1>op2	op1>=op2	op1<=op2	op1==op2	op1!=op2	T	F	op1	op2	op1<=op2	op1!=op2	F		
op1<=op2	op1<op2	op1>op2	op1>=op2	op1<op2	op1==op2	op1!=op2	T	F	op1	op2	op1<op2	op1==op2	T		
op1==op2	op1!=op2	op1>op2	op1>=op2	op1<op2	op1<=op2	op1!=op2	T	F	op1	op2	op1<=op2	op1>=op2	F	op1	op2
op1!=op2	op1==op2	op1>op2	op1>=op2	op1<op2	op1<=op2	op1==op2	T	F	op1	op2	op1<op2	op1>op2	F	op1	op2

Tabla 29: Tabla comparativa operador ROR con otras herramientas.

Es por ello, que se actualiza el operador para contemplar todos y cada uno de los casos que contempla MuJava, excepto los casos de *op1* y *op2*, los cuales serán implementados en

otro operador creado posteriormente, evitando así crear mutantes equivalentes. Con esto se lleva a MuCPP a tener todos y cada uno de los mutantes respecto a este operador. Con esta mejora se multiplica por siete el numero de mutantes generados en cada uso del operador, una mejora de lo más significativa.

En la siguiente tabla puede observarse los mutantes generados tras la actualización por MuCPP:

Código Inicial	**Mutantes en MuCPP Actualizado**						
op1 >op2	op1 >= op2	op1 <op2	op1 <= op2	op1 == op2	op1 != op2	T	F
op1 >= op2	op1 >op2	op1 <op2	op1 <= op2	op1 == op2	op1 != op2	T	F
op1 <op2	op1 >op2	op1 >= op2	op1 <= op2	op1 == op2	op1 != op2	T	F
op1 <= op2	op1 >op2	op1 >= op2	op1 <op2	op1 == op2	op1 != op2	T	F
op1 == op2	op1 >op2	op1 >= op2	op1 <op2	op1 <= op2	op1 != op2	T	F
op1 != op2	op1 >op2	op1 >= op2	op1 <op2	op1 <= op2	op1 == op2	T	F

Tabla 30: Tabla operador ROR tras su actualización.

5.1.8. [COR] Operador de reemplazo de operadores condicionales

Código Inicial	**Mutantes en MuCPP**	**Mutantes en MuJava**						**Mutantes en Major**			
op1 && op2	op1 || op2	op1 || op2	op1 & op2	op1 | op2	op1 op2	op1	op2	op1 == op2	op1	op2	F
op1 || op2	op1 && op2	op1 && op2	op1 & op2	op1 | op2	op1^op2	op1	op2	op1 != op2	op1	op2	T

Tabla 31: Tabla comparativa operador COR con otras herramientas.

Tal y como podemos observar en la tabla anterior, el operador COR se queda muy corto en comparación con la competencia, generando hasta seis veces menos operadores que MuJava y cuatro veces menos que Major, no consiguiendo superar a la competencia. Es por ello que se decide actualizar el operador, implementando todos los casos de MuJava y Major, a excepción de los mutantes *op1* y *op2*, los cuales serán incluidos en otro operador desarrollado posteriormente.

Cabe destacar los mutantes que proporciona Major, dando un *FALSE* en el caso del operador *AND* y un*TRUE* en el caso del operador *OR*, cosa que parece curiosa y que se decide no dejar igual, dando tanto *True* como *False* a ambos operadores, además pasa lo mismo con *==* y *!=*, por lo que se hará lo mismo que en el caso anterior, obteniendo así un operador muy completo y sin rival entre sus principales competidores.

En la siguiente tabla se puede observar como quedará el operador tras la actualización realizada:

Código Inicial	**Mutantes en MuCPP Actualizado**							
op1 && op2	op1 || op2	op1 & op2	op1 | op2	op1^op2	op1 == op2	op1 != op2	T	F
op1 || op2	op1 && op2	op1 & op2	op1 | op2	op1^op2	op1 == op2	op1 != op2	T	F

Tabla 32: Tabla operador COR tras su actualización.

Destacar que con esta mejora el operador COR incrementa de uno a ocho mutantes en cada uso, una mejora muy significativa.

5.1.9. [COD] Operador de eliminación de operadores condicionales

En el caso de este operador de mutación parece interesante su no implementación en la herramienta Major dada sus múltiples actualizaciones constantes. En cambio, en MuJava si se implementa y de la misma forma que MuCPP, por lo que se decide no actualizar el operador de ninguna forma. A continuación se detalla la tabla con el operador en cada herramienta.

Código Inicial	Mutantes en MuCPP	Mutantes en MuJava	Mutantes en Major
!op	op	op	

Tabla 33: Tabla comparativa operador COD con otras herramientas.

5.1.10. [COI] Operador de inserción de operadores condicionales

Al igual que en el apartado anterior, se hace constar la no aparición de este operador en Major, cosa que si pasa, y del mismo modo que en MuCPP, en MuJava. A continuación de puede observar la tabla del operador.

Código Inicial	Mutantes en MuCPP	Mutantes en MuJava	Mutantes en Major
op	!op	!op	

Tabla 34: Tabla comparativa operador COI con otras herramientas.

5.1.11. [LOR] Operador de reemplazo de operadores lógicos

Código Inicial	Mutantes en MuCPP	Mutantes en MuJava						Mutantes en Major	
op1 & op2	op1 | op2	op1 && op2	op1 || op2	op1 | op2	op1 op2	op1	op2	op1 | op2	op1^op2
op1 | op2	op1 & op2	op1 && op2	op1 || op2	op1 & op2	op1^op2	op1	op2	op1 & op2	op1^op2
op1^op2	op1 | op2	op1 && op2	op1 || op2	op1 & op2	op1 | op2	op1	op2	op1 & op2	op1 | op2

Tabla 35: Tabla comparativa operador LOR con otras herramientas.

En este operador en primer lugar nos centraremso en los operadores que MuCPP ya contemplaba antes, como son &, | y ^ , y posteriormente se hablará sobre el que no contempla, $\sim$.

Tal y como podemos observar en la tabla, los tres primeros operadores ya se contemplan en MuCPP, pero no realizan todos los reemplazos que se expresan en otras herramientas, es por ello que se decide actualizar el operador e incluir todas las posibilidades de ambos operadores excepto el *op1* y el *op2*, que se incluirán en un operador diferente.

Tras esta gran actualización del operador, la tabla de mutantes que genera se queda de la siguiente forma:

Código Inicial	**Mutantes en MuCPP**			
op1 & op2	op1 && op2	op1 || op2	op1 | op2	op1 ^ op2
op1 | op2	op1 && op2	op1 || op2	op1 & op2	op1 ^ op2
op1 ^ op2	op1 && op2	op1 || op2	op1 & op2	op1 | op2

Tabla 36: Tabla operador LOR tras su actualización.

5.1.12. [ASR] Operador de reemplazo atajos con asignación

Este operador, denominado ASRS en MuJava, contemplan los casos expuestos en la tabla, y en el caso de Major, no se contempla ningún operador sobre estos operadores, por ello, no se tiene en cuenta en este análisis.

Código Inicial	**Mutantes en MuCPP**	**Mutantes en MuJava**				**Mutantes en Major**
op1 -= op2	op1 += op2	op1 += op2	op1 *= op2	op1 /= op2	op1 %= op2	
op1 += op2	op1 -= op2	op1 -= op2	op1 *= op2	op1 /= op2	op1 %= op2	
op1 *= op2	op1 /= op2	op1 += op2	op1 -= op2	op1 /= op2	op1 %= op2	
op1 /= op2	op1 *= op2	op1 += op2	op1 -= op2	op1 *= op2	op1 %= op2	
op1 %= op2	op1 /= op2	op1 += op2	op1 -= op2	op1 *= op2	op1 /= op2	

Tabla 37: Tabla comparativa operador ASR con otras herramientas.

Observando el operador de MuJava y sus mutantes, los cuales, no son todos contemplados en MuCPP, es por ello, que se actualiza MuCPP insertando estos nuevos mutantes.

Con esta mejora, se insertan tres mutantes nuevos con cada uso de este operador en MuCPP, quedando el operador como queda en la siguiente tabla:

Código Inicial	**Mutantes en MuCPP actualizado**			
op1 -= op2	op1 += op2	op1 *= op2	op1 /= op2	op1 %= op2
op1 += op2	op1 -= op2	op1 *= op2	op1 /= op2	op1 %= op2
op1 *= op2	op1 += op2	op1 -= op2	op1 /= op2	op1 %= op2
op1 /= op2	op1 += op2	op1 -= op2	op1 *= op2	op1 %= op2
op1 %= op2	op1 += op2	op1 -= op2	op1 *= op2	op1 /= op2

Tabla 38: Tabla operador ASR tras su actualización.

5.2. Comparación de operadores de herencia

5.2.1. [IHD] Borrado de variable oculta

Este operador, tal y como indica su nombre, elimina la definición de una variable oculta o sobrescrita por un método hijo. De este modo las referencias a la variable se realizarán

a la clase padre.

Este operador, al igual que en casos anteriores se encuentra tanto en MuCPP como en mujava, y se comportan del mismo modo. Analizan en primer lugar las variables que se encuentran en ambas clases y de ese modo borra la variable de la clase hijo, haciendo que en cada llamada a esa variable se llame a la de la clase padre.

En este ejemplo podemos observar el comportamiento en MuCPP:

```
class Parent{
    int a;
    ...
}
class Child: public Parent{
    int a;
    ...
}
```

Creando un mutante:

```
class Parent{
    int a;
    ...
}
class Child: public Parent{
    //int a;
    ...
}
```

Por otro lado podemos encontrar el comportamiento en MuJava, donde lo único que cambia es la forma de herencia entre clases, pero no el mutante en sí:

```
class Parent{
    int a;
    ...
}
class Child extends Parent{
    int a;
    ...
}
```

Creando un mutante:

```
class Parent{
    int a;
    ...
}
class Child extends Parent{
    //int a;
    ...
}
```

5.2.2. [IHI] Inserción de variable oculta

Este operador, tal y como indica su nombre, inserta la definición de una variable oculta o sobrescrita de una padre en una clase hijo. De este modo las referencias a la variable se realizarán a la clase hijo y no a la clase padre.

Este operador, es justamente el contrario al operador anterior, IHD, y al igual que en ese caso, este operador, se encuentra tanto en MuCPP como en mujava, y se comportan del mismo modo. Analizan en primer lugar las variables que se encuentran solo en la clase padre y de ese modo inserta la variable de la clase hijo, haciendo que en cada llamada a esa variable se llame a la de la clase hijo en lugar de a la padre.

En este ejemplo podemos observar el comportamiento en MuCPP:

```
class Parent{
    int a;
    ...
}
class Child: public Parent{
    ...
}
```

Creando un mutante:

```
class Parent{
    int a;
    ...
}
class Child: public Parent{
    int a;
    ...
}
```

Por otro lado podemos encontrar el comportamiento en MuJava, donde lo único que cambia es la forma de herencia entre clases, pero no el mutante en sí:

```
class Parent{
    int a;
    ...
}
class Child extends Parent{
    ...
}
```

Creando un mutante:

```
class Parent{
    int a;
    ...
}
class Child extends Parent{
    int a;
    ...
}
```

5.2.3. [ISD] Borrado de referencia a clase padre

Mediante este operador, tal y como se indica en su nombre, se elimina la referencia a la clase padre. Este operador se encuentra tanto en las herramientas MuCPP como MuJava, y son idénticos excepto por una diferencia del propio lenguaje. En el caso de C++ se utiliza el operador *::*, en cambio, en Java se utiliza el operador *.* para hacer referencias a otras clases, es por ello, que en MuJava en lugar de buscar el doble simbolo de dos puntos se buscará el simbolo punto y viceversa.

A continuación se puede observar el operador en uso en MuCPP mediante el siguiente ejemplo:

```
class Child: public Parent{
    int metodo(){
        return Parent::a;
    }
    ...
}
```

Creando un mutante:

```
class Child: public Parent{
    int metodo(){
        return a;
    }
    ...
}
```

Y en el caso de MuJava se realizaría de la siguiente forma:

```
class Child extends Parent{
    int metodo(){
        return Parent.a;
    }
    ...
}
```

Creando un mutante:

```
class Child extends Parent{
    int metodo(){
        return a;
    }
    ...
}
```

Como se puede observar son idénticos con la excepción de la forma de heredar y la forma de llamar a los miembros de una clase de cada uno de los lenguajes.

5.2.4. [ISI] Inserción de referencia a clase padre

Mediante este operador, tal y como se indica en su nombre, se inserta la referencia a la clase padre desde una clase hijo cuando se llama a un atributo de la clase. Este operador se encuentra tanto en las herramientas MuCPP como MuJava, y son idénticos excepto por una diferencia del propio lenguaje. En el caso de C++ se utiliza el operador *::*, en cambio, en Java se utiliza el operador . para hacer referencias a otras clases, es por ello, que en MuJava en lugar de buscar el doble simbolo de dos puntos se buscará el simbolo punto y viceversa.

A continuación se puede observar el operador en uso en MuCPP:

```
class Child: public Parent{
    int metodo(){
        return a;
    }
    ...
}
```

Creando un mutante:

```
class Child: public Parent{
    int metodo(){
        return Parent::a;
    }
    ...
}
```

Y en el caso de MuJava se realizaría de la siguiente forma:

```
class Child extends Parent{
    int metodo(){
        return a;
    }
    ...
}
```

Creando un mutante:

```
class Child extends Parent{
    int metodo(){
        return Parent.a;
    }
    ...
}
```

Como se puede observar son idénticos con la excepción de la forma de heredar y la forma de llamar a los miembros de una clase de cada uno de los lenguajes.

Cabe destacar también que es el operador contrario de ISD, es decir, de la eliminación, realizando el trabajo inverso de forma idéntica.

5.2.5. [IOD] Borrado de método sobrescrito

Mediante este operador se simula un fallo del olvido de sobrescribir un método de una clase padre en una clase hija. Este operador, al igual que los anteriores, también se encuentran en MuJava, además de en MuCPP.

Este operador es muy simple y se realiza en ambos lenguajes de la misma forma pero con la excepción de la nomenclatura del propio lenguaje.

En MuCPP el operador se comporta de la siguiente forma:

```
class Child: public Parent {
    name(){...}
}
```

Creando el siguiente mutante:

```
class Child: public Parent {
   // name(){...}
}
```

En el caso de MuJava es idéntico con la excepción de la nomenclatura en la herencia

```
class Child extends Parent {
    name(){...}
}
```

Creando el siguiente mutante:

```
class Child extends Parent {
   // name(){...}
}
```

5.2.6. [IOP] Cambio de posición de la llamada al método sobrescrito

Este operador cambia la posición de la llamada a un método padre desde un método de la clase hijo. Este operador también aparece en ambos lenguajes y de la misma forma.

En MuCPP podemos encontrar el siguiente ejemplo:

```
class Parent {
    void name(){...}
}
class Child: public Parent {
    void name(){
        Parent::name();
        int a;
    }
}
```

Creando el siguiente mutante:

```
class Parent {
    void name(){...}
}
class Child: public Parent {
    void name(){
        int a;
        Parent::name();
    }
}
```

En MuJava podemos encontrar el siguiente ejemplo:

```
class Parent {
    void name(){...}
}
class Child extends Parent {
    void name(){
        Parent.name();
        int a;
    }
}
```

Creando el siguiente mutante:

```
class Parent {
    void name(){...}
}
class Child extends Parent {
    void name(){
        int a;
        Parent.name();
    }
}
```

5.2.7. [IOR] Renombrado método sobrescrito

El cometido de este operador es renombrar el método de la clase padre, de forma, que el método sobrescrito en la clase hijo no afecte a la clase padre de ninguna forma.

Este operador se encuentra tanto en MuCPP como en MuJava y actuan de la misma forma. A continuación podemos ver un ejemplo en cada una de las herramientas. En MuCPP podemos encontrar el siguiente ejemplo:

```
class Parent {
    void name(){...}
}
class Child: public Parent {
    void name(){}
}
```

Creando el siguiente mutante:

```
class Parent {
    void name'(){...}
}
class Child: public Parent {
    void name(){}
}
```

En MuJava podemos encontrar el siguiente ejemplo:

```
class Parent {
    void name(){...}
}
class Child extends Parent {
    void name(){...}
}
```

Creando el siguiente mutante:

```
class Parent {
    void name'(){...}
}
class Child extends Parent {
    void name(){...}
}
```

5.2.8. [IPC] Borrado de llamada explícita al constructor de la clase padre

Este operador, tal y como indica su nombre, elimina la llamada explícita por parte de una clase hijo al constructor de una clase padre.

Cuando un objeto de una clase es inicializado, en primer lugar se llama a los constructores de las clases base de las que hereda. Si no se especifica el constructor de estas clases, se invoca al constructor por defecto. Sin embargo, es posible llamar a un constructor específico. Este operador elimina la llamada explícita a un constructor de la clase padre de manera que sea utilizado el constructor por defecto.

Este operador se encuentra en ambas herramientas, aunque las diferencias existentes son determinadas por ambos lenguajes. A continuación podemos observar un ejemplo en cada una de las herramientas.
En primer lugar podemos observar este ejemplo en el lenguaje C++:

```
class Child: public Parent {
    Child(){
        Parent();
        ...
    }
}
```

Ejemplo en el que MuCPP crea el siguiente Mutante:

```
class Child: public Parent {
    Child(){
        //Parent();
        ...
    }
}
```

En segundo lugar podemos observar este ejemplo en Java:

```
class Child extends Parent {
    Child(){
        super();
    }
}
```

Ejemplo en el que MuJava crea el siguiente Mutante:

```
class Child extends Parent {
    Child(){
        //super();
    }
}
```

5.2.9. [IMR] Reemplazo de herencia múltiple

Este operador que solo es posible encontrarle en MuCPP y no en MuJava, tiene como cometido la modificación o renombrado del la clase a la que se accede para ejecutar un método. Esto sucede cuando una clase hereda de dos o más clases y estas tienen un método en común al que se desea acceder. Por ejemplo, una clase hereda de dos clases A y B, en caso de que se llame al método de la clase A, denominado *metodo* de la forma A::metodo(), el mutante llamará al mismo método de la clase B.

A continuación se plantea un ejemplo para una comprensión más visual. En primer lugar podemos observar este ejemplo en el lenguaje C++:

```
class Parent1 {
    metodo(){...};
}
class Parent2 {
    metodo(){...};
}
class Child: public Parent1, public Parent2 {
    metodo(){
        Parent1::metodo();
    }
}
```

En este ejemplo, MuCPP crearía el siguiente mutante:

```
class Parent1 {
    metodo(){...};
```

```
}
class Parent2 {
    metodo(){...};
}
class Child: public Parent1, public Parent2 {
    metodo(){
        Parent2::metodo();
    }
}
```

En este punto es interesante destacar la no aparición de este operador en MuJava, dado que en Java no existe la herencia múltiple, y por consiguiente, no es posible la creación de este operador.

5.3. Comparación de operadores de polimorfismo y enlace dinámico

5.3.1. [PVI] Inserción del modificador virtual

Mediante este operador se asegura la posibilidad de que el programador olvide incluir en la declaración de un método la palabra virtual, la cual se necesita para emplear el polimorfismo en C++.

En otros lenguajes como Java las funciones son declaradas virtuales por defecto, por lo que no se considera necesario la inclusión de este operador. Es por ello que en herramientas como MuJava no aparecen este tipo de operadores. Cabe destacar que existe la posibilidad de declarar un método como abstracto mediante el modificador *abstract* pero se considera innecesario ya que posee un fin que se asigna por defecto.

En MuCPP esta operador realiza un trabajo muy simple, investigando si un método ha sido declarado tanto en la clase padre como en la clase hijo, y en caso afirmativo y viendo que en la padre no es virtual, se creará el mutante.

Un ejemplo de este operador puede ser el siguiente:

```
class A{
    void C(){...}
}
class B: public A{
    void C(){...}
}
```

En este caso se crean un mutante como sigue:

```
class A{
    virtual void C(){...}
}
class B: public A{
    void C(){...}
}
```

5.3.2. [PCD] Eliminación del operador de modelado de tipos

Este operador realiza la acción contraria a la realizada por la inserción del operador PCI, el cual inserta este mismo operador de modelado.

En MuCPP este operador realiza diversas comparaciones, como si existe ese mismo método en la clase no casteada. En caso afirmativo, se elimina el operador dynamic_cast de la llamada al método.
Este operador en MuCPP se vería de la siguiente forma:

```
Child c;
Parent &p = c;
(dynamic_cast<Child*>(p)).method();
```

En este caso se crea un mutante como sigue:

```
Child c;
Parent &p = c;
p.method();
```

En el caso de otras herramientas como MuJava existe el mismo operador, pero realiza las acciones de otra forma. A continuación podemos observar un ejemplo en MuJava:

```
Child cRef;
Parent pRef = cRef;
((Child) pRef).toString();
```

El cual crearía el siguiente mutante:

```
Child cRef;
Parent pRef = cRef;
pRef.toString();
```

5.3.3. [PCI] Inserción del operador de modelado de tipos

Es el contrario al operador anterior, donde se inserta el modelado de tipos. Esta acción se realiza mediante el operador dynamic_cast en el caso de C++ y mediante un modelado simple entre paréntesis en otros lenguajes como Java. Este operador realiza sus acciones cuando existe un método tanto en la clase padre como en la clase hijo, pudiendo así llamar a métodos de la otra clase mediante el modelado sin cambiar el nombre de esta.

En MuCPP esta operador se vería tal y como sigue, dando como ejemplo lo siguiente:

```
Child c;
Parent &p = c;
p.method();
```

En este caso se crean un mutante como sigue:

```
Child c;
Parent &p = c;
(dynamic_cast<Child*>(p)).method();
```

De la misma forma pasa en herramientas como MuJava, pero con la excepción de lo que se ha explicado anteriormente, en este caso no se realiza mediante un operador como dynamic_cast sino mediante un modelado simple entre paréntesis. A continuación podemos ver un ejemplo de uso en MuJava:

```
Child cRef;
Parent pRef = cRef;
pRef.toString();
```

El cual crearía el siguiente mutante:

```
Child cRef;
Parent pRef = cRef;
((Child) pRef).toString();
```

5.3.4. [PCC] Cambio del tipo de modelado

Mediante este operador no se realiza un modelado ni se elimina uno, sino se modifica el modelado, cambiando la clase a la que se convertirá por otra que contenga un método con el mismo nombre.

En MuCPP, al igual que en casos anteriores, esto se realiza modificando el operador dynamic_cast, tal y como podemos observar en este ejemplo:

```
(dynamic_cast<Child*>(p)).method();
```

En este caso se crean un mutante como sigue:

```
(dynamic_cast<Parent*>(p)).method();
```

En otros lenguajes como Java, al igual que se ha dicho en operadores anteriores, esto se realiza mediante un modelado simple y este operador puede ser encontrado, por ejemplo, en MuJava, realizando la siguiente operación:

```
((Child) pRef).toString();
```

Creando un mutante tal y como sigue:

```
((Parent) pRef).toString();
```

5.3.5. [PMD] Declaración con tipo de clase padre

Mediante este operador se cambiará la declaración de un tipo hijo a un tipo padre. Este operador puede ser encontrado tanto en MuCPP como en su competencia, MuJava.

La acción que realiza este operador es muy simple, ya que, solo modifica una declaración, observando anteriormente que la clase hereda de una clase padre y asignandole dicha clase en la declaración.

Un ejemplo de este operador en ambos lenguajes puede ser el siguiente:

```
Child A;
A = new Child();
```

Creando un mutante:

```
Parent A;
A = new Child();
```

5.3.6. [PPD] Declaración de una variable parámetro con el tipo de una clase hija

Este operador realiza la misma acción que el operador anterior, pero en lugar de declarar la variable como miembro, esta se declara en la llamada a un método o función, a la que se le pasa como parámetro.

Este operador puede ser encontrado tanto en MuJava como en MuCPP, haciendo las mismas acciones en ambas herramientas. Cabe aquí destacar que este operador debe saber cual es la clase padre y si posee algún hijo, por lo que deberá observar todas y cada una de las clases restantes:

Un ejemplo de este operador en ambos lenguajes puede ser el siguiente:

```
metodo(Child a);
```

Creando un mutante:

```
metodo(Parent a);
```

5.3.7. [PNC] Llamada al método *new* con tipo de clase hija

Este operador, el cual también aparece en MuCPP y MuJava, cambia la llamada al método constructor de la clase, llamando al constructor de la clase hijo. Esto simula un posible fallo de programación en el que el programador se ha confundido al realizar la llamada y llama a la clase padre en lugar de a la clase hijo.

Este operador funciona de la misma forma en ambas herramientas y a continuación se puede observar un ejemplo en ambas:

```
Parent A;
A = new Parent();
```

Creando un mutante:

```
Parent A;
A = new Child();
```

5.3.8. [PRV] Asignación de una referencia con otro tipo compatible

Este operador modifica la asignación de una referencia mediante otra con un tipo compatible. Es habitual el error en la asignación de un hijo a una clase padre y equivocarse de hijo, es por ello, que existe este operador, el cual intentará encontrar al hijo correcto creando diversos mutantes entre todos los hijos declarados y compatibles.

Este operador también se encuentra en ambas herramientas, aunque no se modifica de la misma forma. En el caso de MuCPP se vería de la siguiente forma:

```
Parent A;
Child1 B;
Child2 C;
A=&B;
```

Creando un mutante:

```
Parent A;
Child1 B;
Child2 C;
A=&C;
```

En el caso de MuJava no se realizará de la misma forma, ya que no se asignarán direcciones de memoria sino un objeto a otro:

```
Parent A;
Child1 B;
Child2 C;
A=B;
```

Creando un mutante:

```
Parent A;
Child1 B;
Child2 C;
A=C;
```

5.4. Comparación de operadores de sobrecarga

5.4.1. [OMD] Borrado del método sobrecargado

Este operador puede ser encontrado en MuCPP además de en MuJava, como su propio nombre indica, este operador crea mutantes eliminando uno de los métodos sobrecargado con los que cuente la clase. Si el mutante aún sigue vivo tras su ejecución significará que el método al que llama es el incorrecto y por consiguiente se está produciendo una conversión del tipo de parámetro incorrecta.

Un ejemplo de este operador puede ser el siguiente:

```
void metodo( int b ){...}
void metodo( bool b ){...}
```

En este caso se crean dos mutantes:

```
//Mutante 1

//void metodo( int b ){...}
void metodo( bool b ){...}

//Mutante 2

void metodo( int b ){...}
//void metodo( bool b ){...}
```

5.4.2. [OMR] Cambio del contenido del método sobrecargado

Este operador, el cual solo aparece en MuCPP y en MuJava, pero no en Major, tiene como fin el comprobar que se está realizando la llamada al método correcto, por ello se reemplaza el contenido de un método por el contenido de otro con el mismo nombre.

Un ejemplo de este operador puede ser el siguiente:

```
void metodo( int b ){...}
void metodo( bool b ){...}
```

En este caso se crean dos mutantes:

```
//Mutante 1

void metodo( int b ){...}
void metodo( int b, int c ){...}

//Mutante 2

void metodo( int b ){...}
void metodo( int b, int c ){
    this->metodo(b);
}
```

5.4.3. [OAN] Cambio del número de argumentos

Otra manera de asegurar que el programador no invoca el método incorrecto debido a la sobrecarga de métodos es cambiar el número de argumentos en la invocación al método. Análogamente al caso anterior, debe existir la sobrecarga que acepte el cambio en la lista de argumentos.

Para ello se hace uso de los valores predeterminados para los argumentos de C++, es decir, se les asignará valores predeterminados a los argumentos para así poder llamar a los métodos con un numero menor de argumentos, tal y como se puede observar en el ejemplo posterior.

En el caso de Major no se encuentra este operador, en cambio, en MuJava si aparece pero en conjunto con otro operador, denominándo a este conjunto como OAC.

Este operador es implementado en C++ como sigue, dando como ejemplo:

```
s.Push(2, 0.5);
```

En este caso se crean tres mutantes:

```
\\M1
s.Push(2);
\\M2
s.Push(0.5);
\\M3
s.Push();
```

5.4.4. [OAO] Cambio del orden de los argumentos

Este operador modifica el orden de los argumentos, comprobando así que no exista otro método capaz de ejecutar esa llamada, lo cual sería un error de programación. En MuJava existe este operador incluido, tal y como en el caso anterior, en el operador OAC.

Este operador es implementado en MuCPP como sigue, dando como ejemplo:

```
s.Push(2, 0.5);
```

En este caso se crea un mutante:

```
s.Push(0.5, 2);
```

5.5. Comparación de operadores de manejo de excepciones

5.5.1. [EHC] Cambio del manejador de la excepción

En MuCPP este operador crea una excepción en lugar de ser tratada en ese momento, modificando así ese tratamiento y propagando la excepción, para que, con suerte, sea tratada en otro lugar de la ejecución del programa.

Este operador tampoco aparece en las herramientas MuJava o Major aunque, se cree que si deberían contenerlo, dado que es un operador sobre las excepciones, de gran importancia dentro de la programación.

Un ejemplo de este operador puede ser el siguiente:

```
try{
    ...
}catch(){
    ...
};
```

Un mutante creado mediante este operador sería:

```
try{
    ...
}catch(){
    throw;
};
```

5.5.2. [EHR] Eliminación del manejador de excepción

El operador EHR modifica el manejador de la excepción eliminando una de las cláusulas catch, siempre que haya más de una. Al eliminar el manejador, se pospone la captura de la excepción pasándola al manejador más cercano. De esta manera, se comprueba si el programador realiza correctamente el manejo de la excepción.

En otras herramientas como Major o MuJava no se han encontrado operadores con el mismo fin que este.

Un ejemplo de este operador puede ser el siguiente:

```
try{
    ...
}catch(int a){
    ...
}catch(bool b){
    ...
};
```

En este caso se crean dos mutantes:

```
//Mutante 1

try{
    ...
//}catch(int a){
  //  ...
}catch(bool b){
    ...
};

//Mutante 2

try{
    ...
}catch(int a){
    ...
//}catch(bool b){
  //  ...
};
```

5.6. Comparación de operadores de reemplazo

5.6.1. [MCO] Llamada a miembro de otro objeto

El concepto de este operador es muy simple y se basa en el erro de uso de un objeto, en este caso se modifica el objeto al que se realiza la llamada a un método, modificando así a otro objeto de la misma clase.

Este operador no existe en muJava ni en Major, una cosa muy extraña dadas sus múltiples actualizaciones.

Un ejemplo de este operador puede ser el siguiente:

```
Coche a;
Coche b;
a.color();
```

Un mutante creado mediante este operador sería:

```
Coche a;
Coche b;
b.color();
```

5.6.2. [MCI] Llamada a miembro de la clase que se hereda

Este operador es un poco más complicado, dado que se llama a métodos de tipo *virtual*, es decir, métodos de la clase de la que se hereda y se llamará con otro objeto de otra clase pero que heredan de la misma. Este operador tampoco aparecen ni en MuJava ni en Major.

Un ejemplo de este operador puede ser el siguiente:

```
Gato a;
Perro b;
a.color();
```

Un mutante creado mediante este operador sería:

```
Gato a;
Perro b;
b.color();
```

5.7. Comparación de operadores de miscelánea

5.7.1. [CTD] Eliminación de la palabra clave this

El operador de eliminación de la palabra clave this existe solo en MuCPP y MuJava, pero no en Major. Cabe destacar que este operador es de gran importancia dada su finalidad, eliminando la palabra clave *this* en la llamada a expresiones, dejando así llamadas a la clase en caso de que existan, en lugar de llamadas a expresiones del propio método. El uso de punteros *this* puede ser muy común en entornos con gran numero de variables. Las acciones realizadas por este operador en cada herramienta pueden verse en la tabla 39.

Código Inicial	Mutante en MuCPP	Mutante en MuJava
this->op	op	
this.op		op

Tabla 39: Tabla comparativa CTD y JTD en MuCPP y MuJava.

5.7.2. [CTI] Inserción de la palabra clave this

Este operador, al igual que en el ejemplo anterior, aparece en MuCPP y MuJava, pero no en Major. Se considera de gran importancia por el cambio del entorno de las expresiones utilizadas en los diferentes códigos fuente. Las acciones realizadas por este operador en cada herramienta pueden verse en la tabla 40.

Código Inicial	Mutante en MuCPP	Mutante en MuJava
op	this->op	this.op

Tabla 40: Tabla comparativa CTI y JTI en MuCPP y MuJava.

Con este operador se consigue insertar un error muy común en la programación, no haciendo uso del entorno del que se quería hacer uso.

5.7.3. [CID] Eliminación de la inicialización de una variable

Mediante este operador se elimina la inicialización de una variable dada, este es otro de los errores comunes de programación y que se suelen observar habitualmente, dado que el propio compilador asigna una dirección de memoria a la variable, el cual ha podido estar ya asignado a otra variable anteriormente y contener "basura".

En otras herramientas como MuJava este operador posee el nombre de JID. Por otro lado, en Major no consta ningún operador para esta finalidad.

5.7.4. [CDD] Eliminación del destructor de clase

Otro de los fallos de programación habituales son el olvidar la construcción del destructor, dado que en otros lenguajes de programación como Java el recolector de basura realiza este trabajo por nosotros, este olvido es muy habitual. Es por ello que se crea este mutante simulando este error tan común en la programación.

En otras herramientas como MuJava o Major no se encuentra este operador, ya que, como se ha explicado anteriormente, el recolector de basura realiza todo este trabajo por el programador, por lo que no se considera un error de programación el no crear el correspondiente destructor.

5.7.5. [CDC] Creación del constructor por defecto

Es un error muy común no establecer un constructor, dado que existe el constructor por defecto, pero esto suele ser una mala práctica, es por ello que se crea este mutante en caso de que el constructor se haya creado sin ningún parámetro, dejando así el constructor

por defecto y con ello un error de programación muy frecuente.

En otras herramientas como MuJava existe este operador con el nombre de JDC y en Major no existe operadores de este tipo.

5.7.6. [CCA] Eliminación del constructor de copia y del constructor de asignación

Es muy común en el lenguaje de programación C++ la creación del constructor de copia y el constructor de asignación, dado que los constructores por defecto no realizan el trabajo de copia como tal, sino que asignan la dirección de memoria del otro objeto. Es por ello que se crea este mutante, simulando el olvido de la creación del constructor de copia y asignación, haciendo que con cada uso de la copia y asignación se asigne la dirección de memoria y no se copie cada uno de los elementos internos del objeto.

En otros lenguajes de programación como Java no ocurre esto, ya que la asignación copia literalmente todos los elementos de un objeto. Es por esto que las herramientas de pruebas de mutaciones en Java, MuJava y Major no crean este tipo de mutantes y por consiguiente no poseen este operador.

Capítulo 6

Nuevos operadores en MuCPP

Mientras se realizaba la sección anterior, se han conocido nuevos operadores no implementados en la herramienta de prueba de mutaciones MuCPP, pero sí en otras herramientas como MuJava o en la literatura, los cuales han sido expresados y desarrollados en esta sección. Estos operadores son los contenidos en la tabla 41.

Operador
[SOR] Operador de reemplazo de operadores de desplazamiento
[SLD] Eliminación de sentencias
[ODL] Eliminación de operador
[VDL] Eliminación de variables
[CDL] Eliminación de constantes
[CSD] Eliminación de la palabra clave static
[CSI] Inserción de la palabra clave static

Tabla 41: Tabla operadores no implementados hasta la fecha en MuCPP.

A continuación se detallarán cada uno de los operadores, su fin, la acción que realizan y cómo han sido implementados en Clang-8.0, para su posterior importación a la herramienta de prueba de mutaciones.

6.0.1. [SOR] Operador de reemplazo de operadores de desplazamiento

Mediante este operador, el cual, realiza un cambio del operador de desplazamiento, conseguimos incluir los operadores de desplazamiento entre los mutantes de la herramienta, los cuales no habían sido incluidos hasta la fecha dado su poco uso. Con esta inclusión se determina su importancia aunque tenga poco uso, dado que, el que no se use con frecuencia puede llevar al programador a confundir su finalidad o su funcionamiento.

Este operador intercambia los operadores de desplazamiento << por >> y viceversa, además del operador de desplazamiento con asignación <<= por >>= y viceversa, tal y como podemos observar en la tabla siguiente. Esto se debe a que C++ solo proporciona esos cuatro operadores de desplazamiento, en cambio, en MuJava podemos encontrar solo tres

operadores de desplazamiento diferentes e intercambiables, dado que no llevan asignación incluida, por ello se crearán dos mutantes con cada uso de este operador.

Código Inicial	**Mutantes en MuCPP**	**Mutantes en MuJava**	
op1 << op2	op1 >> op2	op1 >> op2	op1 >>>> op2
op1 >> op2	op1 << op2	op1 << op2	op1 >>>> op2
op1 <<= op2	op1 >>= op2		
op1 >>= op2	op1 <<= op2		
op1 >>>> op2		op1 << op2	op1 >> op2

Tabla 42: Tabla mutaciones nuevo operador SOR en MuCPP.

A continuación se expresa la parte del AST Matcher que Clang utilizará para localizar en su árbol la expresión que buscamos, es decir, los operadores de desplazamiento, entre el código fuente de un programa dado.

```
DeclarationMatcher SOR_Matcher(string functionPattern){
    DeclarationMatcher matcher =
    functionDecl(
        matchesName(functionPattern),
        forEachDescendant(
            binaryOperator(
                anyOf(
                    hasOperatorName("<<"),
                    hasOperatorName(">>"),
                    hasOperatorName("<<="),
                    hasOperatorName(">>=")
                )
            ).bind("SOR")
        ),
        unless(anyOf(isImplicitFunction(), isMain())),
     unless(hasAncestor(cxxRecordDecl(isTemplateInstantiation())))
);
    return matcher;
}
```

Tal y como podemos observar se buscarán funciones que contengan en alguno de sus descendientes un operador binario que correspondan con los operadores de desplazamiento o los operadores de desplazamiento con asignación, los cuales son enlazados con el nombre del operador *SOR*. Además se tiene en cuenta que la función sea la misma que la pasado por parámetro, teniendo en cuenta la mejora que se explicará en el punto siguiente.

Por otro lado, se contempla el código fuente de la mutación, el cual es también bastante simple ya que poseemos otros operadores parecidos. Este código es el siguiente:

```
void Mutation::apply_SOR(const MatchFinder::MatchResult &Result){

    if (const BinaryOperator *FS =
        Result.Nodes.getNodeAs<clang::BinaryOperator>("SOR")){
```

```
        Operator = "SOR";

        if( (FullLocation1 =
            checkFullLocation(FS->getBeginLoc())).isValid() ){

            if(checkAction()){

                //Mutate
                r1= SourceRange(FS->getOperatorLoc(),
                    FS->getOperatorLoc().getLocWithOffset(0));
                label = "/*SOR*/";
                switch(FS->getOpcode()){
                    case BO_Shr: label += "<<"; break;
                    case BO_Shl: label += ">>"; break;
                    case BO_ShrAssign: label += "<<="; break;
                    case BO_ShlAssign: label += ">>="; break;
                    default: break;
                };
                replaceText(label, r1);
                createDirectory(1);
                endOperator();
                if(Action == APPLY){
                    applied = true;
                }
            }
        }
    }
}
```

Como se puede observar en esta función, de aplicación del operador SOR sobre el programa fuente, identifica en primer lugar que se trata del operador binario enlazado con el nombre del operador, es decir, *SOR*. Posteriormente se crea el mutante mediante la etiqueta del operador y el caso que corresponda según el operador de desplazamiento o desplazamiento con asignación ante el que nos encontremos. Finalmente se reemplaza el texto mediante la función replaceText() de Clang y LLVM, creando un directorio que guarde el mutante y terminando ese mutante.

Cabe destacar también la forma de localizar la localización del operador que ya se encontraba en el código mediante un SourceRange creado con el comienzo de la localización del operador y el final de este.

La inclusión de este operador en MuCPP es muy sencilla, dado que solo se deben incluir los códigos explicados en diferentes ficheros del programa y la inclusión de distintas lineas para que se ejecuten estas funciones.

6.0.2. [SDL] Eliminación de sentencias

Este operador elimina todas y cada una de las sentencias ejecutables de un código fuente. Cuando se aplica a estructuras de control que incluyen un bloque de declaraciones

(por ejemplo, if, while o for), se elimina todo el bloque, así como cada instrucción interna.

Este operador fue expuesto por Lin Deng, Jeff Offutt y Nan Li en [33], en el que expresan que comenzaron este operador con declaraciones individuales y posteriormente ampliaron a otras estructuras de control. Destacan también que este operador tiene cuatro reglas:

- Deben ser considerados todos los casos posibles.
- Eliminar las condiciones booleanas de las estructuras de control.
- Deben eliminarse las declaraciones internas de las estructuras de control.
- Las estructuras de control anidadas deben tratarse de forma recursiva.

Además de esto se expresa las acciones que se deben realizar sobre todas y cada una de las diferentes estructuras de control más conocidas. Cabe destacar que han sido seguidos todos los parámetros expresados en este artículo para la creación del operador SDL en MuCPP, a excepción de algunos casos que han sido tratados ya en otros operadores anteriormente o casos como la devolución de un verdadero o un falso en el caso de devolver un booleano, que se devolverá un 0 dado que no sabemos de antemano a que tipo nos encontramos, sino solo sabemos si es un objeto o una referencia. Todo esto será explicado posteriormente caso por caso.

En primer lugar veremos el AST Matcher, que, aunque se crea que será muy lioso, no es así, dado que obtendremos todas y cada una de las sentencias del código fuente, o sus diferentes estructuras de control, en otro caso.

```
DeclarationMatcher SDL_Matcher(string functionPattern){
    DeclarationMatcher matcher =
        functionDecl(
            matchesName(functionPattern),
            forEachDescendant(
                compoundStmt(
                    forEach(
                        stmt().bind("SDL")
                    )
                )
            ),
            unless(anyOf(isImplicitFunction(), isMain())),
            unless(hasAncestor(cxxRecordDecl(isTemplateInstantiation()))
        )
    );
    return matcher;
}
```

Tal y como se puede observar, se obtendrán tdos y cada uno de los elementos STMT del AST o, en otras palabras, del código fuente del programa. Un STMT es cualquier estructura de control o sentencia del código fuente, por lo que se marcará todas y cada una de estas estructuras o sentencias mediante el nombre del propio operador, es decir, “SDL”.

Como en los ejemplos anteriores se le pasará el nombre de la función en caso de que se haga uso de la nueva función implementada en el capítulo siguiente.

Por otro lado, podemos observar el código que genera las mutaciones de este operador, el cual es bastante complejo por su extensión.

```
void Mutation::apply_SDL(const MatchFinder::MatchResult &Result){
if (const Stmt *FS = Result.Nodes.getNodeAs<clang::Stmt>("SDL")){
        Operator = "SDL";
        FullSourceLoc FullLocation;
        FullLocation = Context->getFullLoc(FS->getBeginLoc());
        if (FullLocation.isValid() &&
            !Context->getSourceManager().isInSystemHeader(FullLocation)
        ){
            if(checkAction()){
                //Mutation
                if(llvm::isa<WhileStmt>(*FS)){
                    WhileStmt* ws = (WhileStmt*)const_cast<Stmt*>(FS);
                    //While TRUE
                    Rewrite.ReplaceText(SourceRange(
                        ws->getCond()->getBeginLoc(),
                        ws->getCond()->getEndLoc()),
                        "true /*SDL*/"
                    );
                    createDirectory(1);
                    endOperator();
                    resetRewriter();

                    //Eliminacion While completo
                    Rewrite.ReplaceText(SourceRange(
                        ws->getWhileLoc(),
                        ws->getBody()->getEndLoc()),
                        "/*SDL*/"
                    );
                    createDirectory(2);
                    endOperator();
                    resetRewriter();
                }else{
                if(llvm::isa<ForStmt>(*FS)){

                    //Eliminacion for completo
                    ForStmt* fs = (ForStmt*)const_cast<Stmt*>(FS);
                    Rewrite.ReplaceText(SourceRange(
                        fs->getBeginLoc(),
                        fs->getEndLoc()),
                        "/*SDL*/"
                    );
                    createDirectory(1);
                    endOperator();
                    resetRewriter();
```

```
    //Eliminacion de la condicion
    Rewrite.ReplaceText(SourceRange(
        fs->getCond()->getBeginLoc(),
        fs->getCond()->getEndLoc()),
        "/*SDL*/"
    );
    createDirectory(2);
    endOperator();
    resetRewriter();

    //Eliminacion del incremento
    Rewrite.ReplaceText(SourceRange(
        fs->getInc()->getBeginLoc(),
        fs->getInc()->getEndLoc()),
        "/*SDL*/"
    );
    createDirectory(3);
    endOperator();
    resetRewriter();
}else{
if(llvm::isa<IfStmt>(*FS)){
    IfStmt* is = (IfStmt*)const_cast<Stmt*>(FS);
    Rewrite.ReplaceText(
        is->getCond()->getSourceRange(),
        "true /*SDL*/"
    );
    createDirectory(1);
    endOperator();
    resetRewriter();
    Rewrite.ReplaceText(
        is->getCond()->getSourceRange(),
        "false /*SDL*/"
    );
    createDirectory(2);
    endOperator();
    resetRewriter();

    if(is->getElse()){
        Rewrite.ReplaceText(SourceRange(
            is->getBeginLoc(),
            is->getEndLoc()),
            "/*SDL*/"
        );
        createDirectory(3);
        endOperator();
        resetRewriter();
    }
}else{
```

```
if(llvm::isa<SwitchStmt>(*FS)){
    SwitchStmt* ss = (SwitchStmt*)const_cast<Stmt*>(FS);
    const SwitchCase* caso = ss->getSwitchCaseList();
    const SwitchCase* caso2 = caso;
    int iterator = 1;
    Rewrite.ReplaceText(SourceRange(caso->getKeywordLoc(),
        ss->getBody()->getSourceRange().getEnd()),
        "/*SDL*/"+Rewrite.getRewrittenText(
        SourceRange(ss->getBody()->getSourceRange().getEnd(),
            ss->getBody()->getSourceRange().getEnd())
        )+"\n"
    );
    while((caso = caso->getNextSwitchCase())){
        createDirectory(iterator);
        endOperator();
        resetRewriter();
        Rewrite.ReplaceText(SourceRange(
            caso->getKeywordLoc(),
            caso2->getKeywordLoc()),
            "/*SDL*/\n"+Rewrite.getRewrittenText(
                SourceRange(caso2->getKeywordLoc(),
                    caso2->getKeywordLoc())
                )
        );
        caso2=caso;
        iterator++;
    }
}else{
if(llvm::isa<ReturnStmt>(*FS)){
    ReturnStmt* rs = (ReturnStmt*)const_cast<Stmt*>(FS);
    if(!strcmp(rs->getRetValue()->getType()->getTypeClassName(),
    "Builtin")){
        Rewrite.ReplaceText(SourceRange(
            rs->getRetValue()->getExprLoc(),
            rs->getRetValue()->getExprLoc()),
            "/*SDL*/ 0"
        );
    }else{
        Rewrite.ReplaceText(SourceRange(
            rs->getRetValue()->getExprLoc(),
            rs->getRetValue()->getExprLoc()),
            "/*SDL*/ NULL"
        );
    }
    createDirectory(1);
    endOperator();
    resetRewriter();
}else{
if(llvm::isa<SwitchCase>(*FS)){
```

```
                    SwitchCase* ss = (SwitchCase*)const_cast<Stmt*>(FS);
                    Stmt* caso = ss->getSubStmt();
                    Rewrite.ReplaceText(SourceRange(
                        caso->getBeginLoc(),
                        caso->getEndLoc()),
                        "/*SDL*/"
                    );
                    createDirectory(1);
                    endOperator();
                    resetRewriter();
                }else{
                    Rewrite.ReplaceText(SourceRange(
                        FS->getBeginLoc(),
                        FS->getEndLoc()),
                        "/*SDL*/"
                    );
                    createDirectory(1);
                    endOperator();
                    resetRewriter();
                }}}}}}
            }
            if(Action == APPLY){
                applied = true;
            }
        }
    }
}
```

Tal y como podemos observar en el código anterior, este se divide en siete partes totalmente diferentes, conteniendo una dentro del else de otra, haciendo así que no se realicen acciones de un bloque de control y además de una única sentencia. Destacar el uso de llvm::isa para la detección de cada uno de los diferentes bloques de control existentes, además del operador const_cast mediante el que se castea de STMT a los diferentes bloques de control.

En primer lugar podemos observar que se actua sobre los bucles de control while, en los que se crea un mutante poniendo la condición a True, es decir, ejecutando infinitamente el bloque de control y otro eliminando el bloque while al completo. Aquí podemos ver un ejemplo de los mutantes que se crean sobre este bloque:

Código Inicial	**Mutantes en MuCPP**	

...
while(x!= 0){
x=x+1;
}
...
	...
	while(true){
	x=x+1;
	}
	...
		...
		...

Tabla 43: Tabla operador SDL sobre bloques while.

En segundo lugar podemos observar como se actua sobre los bloques en los que se elimina en primer lugar el bloque completo, posteriormente se crea otro mutante eliminando la condición y, por último, se crea un mutante eliminando el incremento/decremento. A continuación podemos observar en la tabla los diferentes mutantes que se crearían con este operador.

Código Inicial	**Mutantes en MuCPP**		

...
for(int i=0; i<10; i++){
x=x+1;
}
...
	...
	...
		...
		for(int i=0; ; i++){
		x=x+1;
		}
		...
			...
			for(int i=0; i<10;){
			x=x+1;
			}
			...

Tabla 44: Tabla operador SDL sobre bloques for.

En tercer lugar encontramos los bloques if, los cuales, son un poco diferentes en el comparación con los mutantes generados en el artículo explicado anteriormente. Esto se debe a que tal y como indican las reglas se deben coger todos y cada uno de los caminos. Por ello, en primer lugar, se pondrá la condición a verdadero, haciendo que pase por la parte del verdadero si o si. Por otro lado se pondrá a falsa, haciendo que si existe el else, pase por la parte esa parte y en caso contrario esto hará que no entre en la condición if, haciendo como si hubiera sido eliminada, por lo que no se borrará el bloque. En caso de que exista el else, también se borrarán ambos bloques, haciendo así que no pase por ninguno de los dos. A continuación puede observar este caso en la siguiente tabla.

Código Inicial	**Mutantes en MuCPP**		

...
if(x=0){
...
}else{
...
}
...
	...
	if(true){
	...
	}else{
	...
	}
	...
		...
		if(fase){
		...
		}else{
		...
		}
		...
			...
			...

Tabla 45: Tabla operador SDL sobre bloques if-else.

En cuarto lugar se ocupará de los bloques switch, en los que se eliminarán uno a uno cada uno de los casos que este contenga. Eliminando así el caso completo y no cada una de las sentencias que este contenga.

En quinto lugar se realizan las devoluciones de las funciones, es decir, los *return*, haciendo que se devuelva cero en caso de que nos encontremos que se está devolviendo un *Builtin* y *NULL* en caso contrario, con esto conseguimos que si nos encontramos ante una dirección de memoria se devuelva NULL y en cualquier otro caso se devuelva un cero. En [39] se puede observar que los tipos Builtin son todos los básicos, es decir, int, char, float, bool, etc. Es por ello que se usa esta técnica dado que es considerada más sencilla a la hora de programar el código. A continuación se pueden observar cada uno de los tipos y lo que devuelve MuCPP cuando crea el mutante.

Tipo del return	Mutantes en MuCPP
int	return 0;
char	return 0;
bool	return 0;
float	return 0;
double	return 0;
array	return NULL;
puntero	return NULL;
referencia	return NULL;
clase	return NULL;
estructura	return NULL;
union	return NULL;
enumeración	return NULL;
typedef	return NULL;

Tabla 46: Tabla operador SDL sobre returns.

En sexto lugar se examina si nos encontramos ante un SwitchCase, ante los cuales, se eliminará la primera sentencia, pero no el "case –:", como si se realizaría si no tuviéramos esta parte.

Y en séptimo y ultimo lugar se examinan las sentencias individuales, ya sean internas a un bloque o no, ante las cuales se elimina la sentencia al completo. Es por ello, que en cada uno de los bloques anteriores no se ha expresado la eliminación de cada una de las sentencias internas.

Destacar que la implementación de este operador ha sido desarrollada en gran medida en base a lo estipulado en el artículo descrito al principio de este operador.

6.0.3. [ODL] Eliminación de operador

Mediante este operador se elimina cualquier operador existente en C++, ya sea aritmético, relacional, lógico, de bit a bit o de asignación. Cuando es el caso de la eliminación de un operador de asignación (por ejemplo, "a + = 2") simplemente se elimina el operador y se sustituye por una asignación simple ("a = 2"). En el caso del operador binario también se debe eliminar un operando para que la expresión esté correcta para poder ser compilada. Es por ello que al eliminar un operador binario se crean dos mutantes, uno cuando se elimina el operando derecho y otro cuando se elimina el operando izquierdo.

Este operador se considera de gran importancia dado que se modificarán todos y cada uno de los operadores, dejando uno de los operandos en el caso de los operadores binarios y una asignación en el caso de los operadores de asignación. Con ello se consigue comprobar si el desarrollador se ha equivocado a la hora de realizar cualquier tipo de operación, proponiendo todas y cada una de las opciones.

Este operador es expuesto por Delamaro, Jeff Offutt y Paul Ammann en [34] en el cual indican que el objetivo de este operador es el intento de mejorar el operador SDL, el cual ya ha sido explicado anteriormente. También se expresa que es lógico esperar que este operador no cree mutantes equivalentes, aunque puede existir algunos mutantes equivalentes

que no indican un error, un ejemplo de ello es “if(x!=0)”, que creará el mutante “if(x)”, lo cual será un mutante que a efectos prácticos hace lo mismo pero no se considera un mutante equivalente dado que ambos programas no son iguales. Por último se indica que a veces si un mutante ODL es equivalente depende de aspectos dinámicos del programa y detectarlos requiere demasiado análisis ya sea para un ser humano o un algoritmo, e incluso puede ser indecidible en algunas circunstancias.

Tras este artículo se implementan en herramientas como MuJava, la cual lo realiza de manera exactamente igual a la que se ha implementado para nuestra herramienta, MuCPP. A continuación se expresa el AST Matcher que utilizará Clang para encontrar las expresiones binarias que deseamos localizar para este operador.

```
DeclarationMatcher ODL_Matcher (string functionPattern) {
    DeclarationMatcher matcher =
        functionDecl(
            matchesName(functionPattern),
            anyOf(
                forEachDescendant(
                    binaryOperator(
                        anyOf(
                            hasOperatorName("+"),
                            hasOperatorName("-"),
                            hasOperatorName("*"),
                            hasOperatorName("/"),
                            hasOperatorName("%"),
                            hasOperatorName("<"),
                            hasOperatorName(">"),
                            hasOperatorName("<="),
                            hasOperatorName(">="),
                            hasOperatorName("=="),
                            hasOperatorName("!="),
                            hasOperatorName("&&"),
                            hasOperatorName("&"),
                            hasOperatorName("||"),
                            hasOperatorName("|"),
                            hasOperatorName("^"),
                            hasOperatorName("<<"),
                            hasOperatorName(">>"),
                            hasOperatorName("+="),
                            hasOperatorName("-="),
                            hasOperatorName("*="),
                            hasOperatorName("/="),
                            hasOperatorName("%="),
                            hasOperatorName("<<="),
                            hasOperatorName(">>="),
                            hasOperatorName("&="),
                            hasOperatorName("|="),
                            hasOperatorName("^=")
                        )
                    ).bind("ODL")
```

```
                )
            ),
            unless(anyOf(isImplicitFunction(), isMain())),
            unless(hasAncestor(cxxRecordDecl(isTemplateInstantiation()))
        )
    );
    return matcher;
}
```

Tal y como se ha podido observar en este AST Matcher, existen dos partes muy diferenciadas. En primer lugar una parte que obtiene las partes del árbol generado por Clang con operadores binarios sin asignación. Para esta primera parte se han puesto todos y cada uno de los operadores sin asignación, sean del tipo que sean. Estos operadores son marcados mediante el nombre del operador y el numero uno ("ODL1") para así poder identificarlos posteriormente. En segundo lugar podemos observar todos y cada uno de los operadores binarios de C++ que contengan una asignación, y al igual que el primero, este es marcado, pero con un dos ("ODL2").

Destacar el uso de nuevo de *matchesName(functionPattern)*, mediante el cual se realizarán los marcados solo en la función o método específico que se le pase por parámetro a la función ODL_Matcher.

Se puede observar también que no se contempla la inclusión de ningún operador unario (como por ejemplo el lógico), esto se debe a que este tipo de operadores no son nuestro objetivo dado que otros operadores se encargan de generar esos mismos mutantes, como puede ser, por ejemplo, el operador LOD.

A continuación podemos observar el código de generación de mutantes de este operador de mutación. En el se pueden observar dos partes bien diferenciadas igual que en el AST Matcher.

```
void Mutation::apply_ODL(const MatchFinder::MatchResult &Result){
    if (const BinaryOperator *FS =
    Result.Nodes.getNodeAs<clang::BinaryOperator>("ODL")){
        Operator = "ODL";
        FullSourceLoc FullLocation;
        FullLocation = Context->getFullLoc(FS->getBeginLoc());
        string oper = Rewrite.getRewrittenText(FS->getOperatorLoc();
        if((oper.compare("+=")&&oper.compare("-=") &&
        oper.compare("*=") && oper.compare("/=") &&
        oper.compare("%=") && oper.compare("<<=") &&
        oper.compare(">>=") && oper.compare("&=") &&
        oper.compare("|=") && oper.compare("^=")){
            if (FullLocation.isValid() &&
            !Context->getSourceManager().isInSystemHeader(FullLocation)){
                attribute_variable += 2;
                if(checkAction()){
                //MUTATION LEFT EXPRESSION
                operators[Operator]->incrementNumberOperands();
                Rewrite.ReplaceText(FS->getLHS()->getSourceRange(),"");
```

```
                Rewrite.ReplaceText(FS->getOperatorLoc(), "/*ODL*/");
                createDirectory(1);
                endOperator();
                resetRewriter();

                //MUTATION RIGHT EXPRESSION
                Rewrite.ReplaceText(FS->getRHS()->getSourceRange(),"");
                Rewrite.ReplaceText(FS->getOperatorLoc(), "/*ODL*/");
                createDirectory(2);
                endOperator();
                resetRewriter();

                if(Action == APPLY){
                applied = true;
                }}
            }
        }else{
            if (FullLocation.isValid() &&
            !Context->getSourceManager().isInSystemHeader(FullLocation)){
                if(checkAction()){
                    //Mutación
                    Rewrite.ReplaceText(FS->getOperatorLoc(), "/*ODL*/ =");
                    createDirectory(1);
                    endOperator();
                    resetRewriter();

                    if(Action == APPLY){
                        applied = true;
                    }
                }
            }
        }
    }
}
```

Tal y como podemos observar, en la primera parte se encargará de los operadores sin asignación, en los cuales genera dos mutantes, aquel que solo deja el operando izquierdo y otro que solo deja el operando derecho. Cabe destacar que aunque uno de esos dos operandos contenga otra operación, se realizan como si se encontraran entre paréntesis, dada la propia estructura de los AST, tal y como vimos en su apartado correspondiente.

Por otro lado, en la segunda parte, podemos observar como se encarga de los operadores de asignación, de los cuales no elimina ningún operando, sino que cambia el operador que posea por una asignación simple. Esto se debe a que si eliminámaramos algún operando se quedaría con un solo operando y no se realizaría ninguna acción. A continuación podemos observar una tabla con las acciones que realiza este operador.

Código Inicial	**Mutantes en MuCPP**	
op1 + op2	op1	op2
op1 - op2	op1	op2
op1 * op2	op1	op2
op1 / op2	op1	op2
op1 % op2	op1	op2
op1 <op2	op1	op2
op1 >op2	op1	op2
op1 <= op2	op1	op2
op1 >= op2	op1	op2
op1 == op2	op1	op2
op1 != op2	op1	op2
op1 && op2	op1	op2
op1 & op2	op1	op2
op1 || op2	op1	op2
op1 | op2	op1	op2
op1 ^op2	op1	op2
op1 <<op2	op1	op2
op1 >>op2	op1	op2
op1 += op2	op1 = op2	
op1 -= op2	op1 = op2	
op1 *= op2	op1 = op2	
op1 /= op2	op1 = op2	
op1 %= op2	op1 = op2	
op1 <<= op2	op1 = op2	
op1 >>= op2	op1 = op2	
op1 &= op2	op1 = op2	
op1 —= op2	op1 = op2	
op1 ^= op2	op1 = op2	

Tabla 47: Tabla operador ODL en MuCPP.

Tal y como podemos observar en la tabla, se crean dos mutantes en el caso de los operadores binarios con asignación y uno en el caso que contiene asignación.

Un caso de especial atención en este operador es la creación de mutantes en los operadores binarios sin asignación que contiene otra operación dentro de uno de sus operandos. Este es el caso del ejemplo que se puso en su momento en la sección del AST y que ahora recordaremos. En ese ejemplo se usó la operación a=1+2+3;, obteniendo el siguiente árbol.

```
-CompoundStmt 0x55b9ea06c1e0 <col:39, line:6:1>
    |-DeclStmt 0x55b9ea06c190 <line:4:2, col:13>
    | `-VarDecl 0x55b9ea06c080 <col:2, col:12> col:6 a 'int' cinit
    |   `-BinaryOperator 0x55b9ea06c168 <col:8, col:12> 'int' '+'
    |     |-BinaryOperator 0x55b9ea06c120 <col:8, col:10> 'int' '+'
    |     |  |-IntegerLiteral 0x55b9ea06c0e0 <col:8> 'int' 1
    |     |  `-IntegerLiteral 0x55b9ea06c100 <col:10> 'int' 2
    |     `-IntegerLiteral 0x55b9ea06c148 <col:12> 'int' 3
```

Observamos usando este operador que se crean cuatro mutantes con el siguiente orden:

- 3;
- 1+2;
- 2+3;
- 1+3;

Tal y como podemos ver se crea en ese orden por una razón. En primer lugar se crea el mutante con el operador + de la derecha, dado que es el que tenemos más externamente en el árbol. Eliminando así, en primer lugar el operando de la derecha, es decir, toda la rama más interna a ese operador (1+2) y en segundo lugar se elimina el operando de la derecha, es decir, la hoja del árbol que contiene el número tres. Posteriormente se pasará al operador suma más interno, eliminando en primer lugar su operando izquierdo, en este caso la hoja que contiene el número uno y posteriormente otro mutante eliminando el operador derecho, en este caso, la hoja con el número dos.

6.0.4. [VDL] Eliminación de variables

En este operador las apariciones de variables se eliminan de cada expresión. Cuando es necesario para compilación, los operadores también se eliminan.

Este operador se localiza en la herramienta MuJava y en diferentes artículos de investigación como es [34], pero se determina que no es necesario, ya que, tal y como expresa el propio autor, los mutantes generador por este operador ya se realizan mediante el operador SDL, el cual, elimina todas y cada una de las sentencias contenidas en el código.

Destacar que su finalidad, según el autor es la de consumir menos recursos en caso de que no se deseen crear tantos mutantes y solo se necesite crear mutantes de las diferentes variables encontradas, lo cual, no se considera de importancia.

6.0.5. [CDL] Eliminación de constantes

En este operador las apariciones de constantes se eliminan de cada expresión. Cuando es necesario para compilación, los operadores también se eliminan.

Este operador, al igual que el anterior, se localiza en la herramienta MuJava y en diferentes artículos de investigación como es [34], pero se determina que no es necesario, ya que, tal y como expresa el propio autor, los mutantes generador por este operador ya se realizan mediante el operador SDL, al igual que en el caso anterior.

Destacar que su finalidad, según el autor es la misma que en el operador anterior y se determina que no es tan relevante para el objetivo de este proyecto.

6.0.6. [CSD] Eliminación de la palabra clave static

Dado que C++ en sus versiones anteriores a la del año 2017 no permitía la inicialización de una variable static en la misma línea y se debía realizar fuera de la clase, no se había creado este operador por los problemas que esto podía acarrear, dado que no se sabría

donde se inicializaba esa variable para poder eliminarlo. Es por ello, que, dado que en la versión c++17 se incluyó la posibilidad de inicialización de las variables static en la misma línea mediante la clausula inline, se crea este operador.

Cabe destacar que este operador solo se utiliza en caso de que la sentencia sea de la forma *inline static tipo var = valor;*, dado que no se podría realizar de otra forma dado que no sabríamos donde buscar la inicialización de dicha variable. A continuación se presenta el AST Matcher que usará Clang para encontrar las líneas ante las que podríamos actuar con este operador.

```
DeclarationMatcher CSD_Matcher(string classPattern){
    DeclarationMatcher matcher = varDecl(
        isStaticStorageClass(),
        matchesName(classPattern)
    ).bind("CSD");
    return matcher;
}
```

Tal y como podemos observar, buscará las variables estáticas dentro de las diferentes clases y se marcarán con el nombre del operador, *CSD*. Destacar también el uso, como en los casos anteriores, del *matchesName(classPattern)*, que hará posible la utilización de una de las mejoras espuestas en el capítulo siguiente.

Por otro lado, podemos observar el código que generará los mutantes de este operador:

```
void Mutation::apply_CSD(const MatchFinder::MatchResult &Result){
    if (const VarDecl *FS =
        Result.Nodes.getNodeAs<clang::VarDecl>("CSD")
    ){
        Operator = "CSD";
        FullSourceLoc FullLocation;
        FullLocation = Context->getFullLoc(FS->getBeginLoc());
        if (FullLocation.isValid() &&
            !Context->getSourceManager().isInSystemHeader(FullLocation)
        ){
        if(
        Rewrite.getRewrittenText(
            FS->getSourceRange()).find("inline")
                !=Rewrite.getRewrittenText(FS->getSourceRange()).npos
        ){
        if(Rewrite.getRewrittenText(FS->getSourceRange()).find("static")!=
            Rewrite.getRewrittenText(FS->getSourceRange()).npos
        ){
            if(checkAction()){
                //Mutation
                string s = Rewrite.getRewrittenText(FS->getSourceRange());
                s.replace(s.find("inline"),6,"");
                s.replace(s.find("static"),6,"/*CSD*/");
                replaceText(s, FS->getSourceRange());
                createDirectory(1);
                endOperator();
                resetRewriter();
```

```
                if(Action == APPLY){
                    applied = true;
                }}}
            }
        }
    }
}
```

Tal y como se puede observar en el operador, se revisa en primer lugar que sea un *inline static* y, posteriormente, se eliminan ambas palabras mediante una búsqueda, creando posteriormente el mutante correspondiente. Cabe destacar el uso de la función *getRewrittenText* la cual no había sido utilizada hasta el momento y obtiene el texto de la sentencia completa. En la siguiente tabla se puede observar la acción que realiza este operador.

Código Inicial	**Mutantes en MuCPP**
inline static int a = 5;	int a = 5;

Tabla 48: Tabla operador CSD en MuCPP.

6.0.7. [CSI] Inserción de la palabra clave static

Tal y como se ha explicado anteriormente C++ en sus versiones anteriores a la del año 2017 no permitía la inicialización de una variable static en la misma línea y se debía realizar fuera de la clase, no se había creado este operador por los problemas que esto pudíera acarrear, dado que no se sabría donde inicializar las diferentes variables. Es por ello, que, dado que en la versión c++17 se incluyó la posibilidad de inicialización de las variables static en la misma linea mediante la clausula inline, se crea ahora este operador.

Cabe destacar que este operador solo se utiliza en caso de que la sentencia sea de la forma *tipo var = valor;*, ya que se desea que la variable esté inicializada en el momento ya que no se podría inicializar posteriormente en ningún otro lado. A continuación se presenta el AST Matcher que usará Clang para encontrar las lineas ante las que podríamos actuar con este operador.

```
DeclarationMatcher CSI_Matcher(string classPattern){
    DeclarationMatcher matcher = declaratorDecl(
        matchesName(classPattern),
        fieldDecl().bind("CSI")
    );
    return matcher;
}
```

Tal y como podemos observar, buscará las variables dentro de las diferentes clases y se marcarán con el nombre del operador, *CSI*. Destacar también el uso, como en los casos anteriores, del *matchesName(classPattern)*, que hará posible la utilización de una de las mejoras expuestas en el capítulo siguiente.

Por otro lado, podemos observar el código que generará los mutantes de este operador:

```
void Mutation::apply_CSI(const MatchFinder::MatchResult &Result){
    if (const Decl *FS = Result.Nodes.getNodeAs<clang::Decl>("CSI")){
    Operator = "CSI";
    FullSourceLoc FullLocation;
    FullLocation = Context->getFullLoc(FS->getBeginLoc());
    if (FullLocation.isValid() &&
        !Context->getSourceManager().isInSystemHeader(FullLocation)
    ){
        if(FS->isDefinedOutsideFunctionOrMethod() && (
            Rewrite.getRewrittenText(
                FS->getSourceRange()).find("=")!=Rewrite.getRewrittenText(
                FS->getSourceRange()).npos)
            ){
            if(checkAction()){
                Rewrite.InsertTextAfter(
                    FS->getSourceRange().getBegin(),
                    "/* CSI */ inline static "
                );
                createDirectory(1);
                endOperator();
                resetRewriter();
                if(Action == APPLY){
                applied = true;
                }}
            }
        }
    }
}
```

En este código podemos observar como se revisa en primer lugar que esté definido fuera de cualquier función o método y que en su contenido posea un igual, es decir, una asignación. Posteriormente, se insertarán tanto la palabra *inline* como *static*, creando el mutante correspondiente. En la siguiente tabla se puede observar la acción que realiza este operador.

Código Inicial	**Mutantes en MuCPP**
int a = 5;	inline static int a = 5;

Tabla 49: Tabla operador CSI en MuCPP.

Capítulo 7

Mejoras en MuCPP

Las mejoras que se han realizado en el presente Trabajo Fin de Grado sobre la herramienta de prueba de mutaciones MuCPP se basan en la inclusión de nuevas funcionalidades y la actualización del frontend. Estas mejoras son expuestas en la tabla 50 y desarrolladas en las siguientes secciones.

Secciones
Inclusión de nuevas funcionalidades.
MuCPP centrado en clases
MuCPP centrado en métodos y funciones
Actualización del frontend
Actualización a Clang-6.0
Actualización a Clang-8

Tabla 50: Tabla mejoras en MuCPP.

7.1. Inclusión de nuevas funcionalidades.

Tras haber actualizado y creado los mutantes que se han considerado indispensables para la herramienta que poseemos, se ha dado comienzo a la inclusión de nuevas funcionalidades para mejorar su funcionamiento.

Dado que en las grandes empresas de cualquier sector nos encontramos ante códigos de gran longitud con centenares o incluso miles de ficheros y líneas de código, donde podemos observar múltiples funciones y/o clases, ponemos en cuestión el uso de MuCPP en estos programas. MuCPP crea todos y cada uno de los mutantes que encuentra en el código completo, por lo que si encontramos códigos tan extensos, la creación de mutantes tendrá una larga duración, por ello se ha considerado necesario la inclusión de dos nuevas funcionalidades implementadas como parámetros con los que limitar el número de mutantes a crear, determinando la clase y/o la función específica de la que se desea obtener los mutantes.

Ambas funcionalidades, se crean con la intención de centrarse en partes concretas de un código más extenso, cosa que será de gran ayuda a la hora de utilizar la herramienta

en la Industria 4.0, lo cual es uno de los fines del presente proyecto.

Las funcionalidades que se integran son las siguientes:

- Si el usuario inserta –class=“ NombreClase ”, la ejecución de MuCPP se realiza sobre la clase.
- Si el usuario inserta –function=“ NombreFuncion ”, la ejecución de MuCPP se realiza sobre la función/método.

Cabe destacar que antes de incluir estas funcionalidades, se observa que no se puede compilar MuCPP a no ser que nos encontremos ante una máquina virtual que contenga Ubuntu 14.04 y Clang-3.6. Posterior a la implementación de estas mejoras ante las que nos encontramos, se actualizará a una versión más reciente del compilador Clang++.

Para esta mejora se han tenido que actualizar todos y cada uno de los patrones asociados a los diferentes operadores de mutación, incluyendo, **matchesName(classPattern)** en el caso de que nos encontráramos sobre un operador de mutación referente a clase, y **matchesName(functionPattern)** en el caso de que nos encontráramos ante una función externa o ante un método de una clase. También, una vez modificados los mutantes, se dividen en tres grupos haciendo más sencillo su control a la hora del paso de parámetros y a la hora de ejecutarlos, siendo los siguientes grupos:

- Mutantes sobre clases.
- Mutantes sobre métodos de clases.
- Mutantes sobre funciones externas a clases.

7.1.1. MuCPP centrado en clases

Dado que C++ es un lenguaje orientado a objetos, se presupone que los desarrolladores de este lenguaje harán un gran uso de clases, y métodos dentro de cada una de ellas. Es por ello que incluimos esta nueva funcionalidad mediante la que el desarrollador podrá comprobar que las pruebas sobre una clase concreta que ha implementado se realizan correctamente.

Esta funcionalidad podrá utilizarse de dos formas distintas:

- Haciendo uso del símbolo “*” mediante –class=“*”, se crearán los mutantes de todas las clases.
- Haciendo uso del nombre de la clase mediante –class=“NombreClase”, se crearán los mutantes de la clase NombreClase.

Esta nueva funcionalidad se ha creado mediante la inclusión de **matchesName(classPattern)** en todos y cada uno de los patrones de los operadores de mutación referidos a clases que contempla MuCPP. Con ello, el propio AST de Clang testea que nos encontremos sobre la clase que se le ha pasado por parámetro.

7.1.2. MuCPP centrado en métodos y funciones

Se presupone que como C++ es utilizado tanto con métodos como con funciones, se decide diferenciar la funcionalidad sobre clases y sus métodos y sobre funciones. Es por ello que incluimos esta nueva funcionalidad mediante la que el desarrollador podrá comprobar que las pruebas sobre una función/método concreto que ha implementado se realizan correctamente.

Esta funcionalidad podrá utilizarse de tres formas distintas:

- Haciendo uso del símbolo "*" mediante –function="*", se crearán los mutantes de todas las funciones/métodos.
- Haciendo uso del nombre de la función mediante –function="NombreFuncion", se crearán los mutantes de todas las clases con ese nombre de función.
- Haciendo uso del nombre del método mediante –function="NombreClase:NombreMetodo", se crearán los mutantes del método NombreMetodo de la clase NombreClase.

Esta nueva funcionalidad se ha creado mediante la inclusión de **matchesName(functionPattern)** en todos y cada uno de los mutantes referedos a funciones o métodos que contempla MuCPP. Con ello, el propio AST de Clang testea que nos encontremos sobre el método, y clase en algunos cases, que se le ha pasado por parámetro.

7.2. Actualización del frontend

En este capítulo se explicarán las dos grandes actualizaciones que se han realizado sobre el frontend Clang de MuCPP, pasando en la primera actualización de Clang-3.6 a Clang-6.0 y en la segunda de Clang-6.0 a Clang-8.

7.2.1. Actualización a Clang-6.0

Para llevar a cabo la presente actualización, se crea una máquina virtual nueva, instalando Ubuntu 18.04 LTS, Clang-6.0, libclang-6.0-dev y make, todo ello mediante la siguiente sentencia:

```
sudo apt install clang-6.0 libclang-6.0-dev make
```

Posteriormente, dado que el Makefile para compilar MuCPP utiliza directrices denominando a Clang-version como Clang, Clang++-version como Clang++ y llvm-config-version como llvm-config, se crean los enlaces simbólicos correspondientes mediante las siguientes sentencias:

```
sudo ln -s /usr/bin/clang-6.0 /usr/bin/clang
sudo ln -s /usr/bin/clang++-6.0 /usr/bin/clang++
sudo ln -s /usr/bin/llvm-config-6.0 /usr/bin/llvm-config
```

Una vez realizadas todas estas tareas, se intenta compilar MuCPP, aportando diversos errores, en primer lugar se encuentran diversos errores en el fichero **mutationsmatchers.cpp**, debido a que en las nuevas versiones se han modificado las denominaciones de múltiples clases y métodos, por ello se modifica el fichero realizando los siguientes cambios:

Original	Cambio
recordDecl	cxxRecordDecl
constructorDecl	cxxConstructorDecl
destructorDecl	cxxDestructorDecl
methodDecl	cxxMethodDecl
constructExpr	cxxConstructExpr
newExpr	cxxNewExpr
memberCallExpr	cxxMemberCallExpr
thisExpr().	cxxThisExpr().
catchStmt	cxxCatchStmt
tryStmt	cxxTryStmt
isCopyAssignmentOperator	CopyAssignOperator
isCopyConstructor	CopyConstructor

Tabla 51: Tabla de cambios en el fichero mutationmatchers.cpp de MuCPP para actualización a Clang-6.0.

Posteriormente, se encuentran diversos errores en la función **collect** del fichero **mutation.cpp**, la cual es la siguiente:

```
bool Mutation::collect(const CXXRecordDecl *Base, void *OpaqueData){
    Mutation *Data = reinterpret_cast<Mutation*>(OpaqueData);
    if(Data->checkClassLocation(Base))
        Data->Bases.insert(Base);
    return true;
}
```

Para arreglar el presente fallo se decide crear un método nuevo con la misma funcionalidad que el anterior pero con diferente denominación y actualizando los errores que se encontraran, eliminando la anterior y quedando definitivamente de la siguiente forma:

```
bool Mutation::FAB(const CXXRecordDecl *Base, llvm::SmallPtrSet<const CXXRecordDecl*, 4> &Bases)
    auto collect = [&] (const CXXRecordDecl *Base){
        if(this->checkClassLocation(Base))
            Bases.insert(Base);
            return true;
    };
    return Base->forallBases(collect, true);
}
```

Una vez arreglados todos los errores que se generaban al intentar compilar MuCPP, pasamos a los *warnings*, entre ellos encontramos diferentes advertencias debidas al makefile, las cuales son arregladas con los siguientes cambios:

Tras esto podemos observar otras advertencias en el archivo **MuCPP.cpp**, en las que se pueden observar diversas conversiones entre modalidades de enteros, concretamente *int* y *unsigned*, donde una vez realizados los *modelados* correspondientes se consigue que no se muestren más advertencias.

Tras todos estos cambios se compila sin problemas MuCPP y su uso es idéntico, por

Original	Cambio
	LLVMCXXFLAGS:= -I/usr/lib/llvm-6.0/include -std=c++0x -fPIC -fvisibility-inlines-hidden\-Werror=date-time -std=c++11 -Wall -W -Wno-unused-parameter -Wwrite-strings -Wcast-qual -Wno-missing-field-initializers -pedantic -Wno-long-long -Wno-uninitialized -Wdelete-non-virtual-dtor -Wno-comment -ffunction-sections -fdata-sections -O2 -DNDEBUG -fno-exceptions -D_GNU_SOURCE -D__STDC_CONSTANT_MACROS -D__STDC_FORMAT_MACROS -D__STDC_LIMIT_MACROS
CXXFLAGS:= $(shell llvm-config –cxxflags) $(RTTIFLAG) -fexceptions	CXXFLAGS:= $(LLVMCXXFLAGS) $(RTTIFLAG) -fexceptions

Tabla 52: Tabla de cambios en el makefile de MuCPP para actualización a Clang-6.0.

lo que se puede afirmar que MuCPP ha sido actualizado a clang-6.0, **cabe destacar que para la gran actualización que suponía pasar de *Clang-3.6* a *Clang-6.0*, no se han tenido que realizar una cantidad de cambios considerables**.

7.2.2. Actualización a Clang-8

Dado que poco tiempo antes de la presentación del presente proyecto aparecen dos nuevas versiones de Clang disponibles en Ubuntu 18.04.3, se decide actualizar la herramienta a la ultima versión disponible, la cual es Clang-8, además de Llvm-8. Para llevar a cabo la presente actualización, se crea una máquina virtual nueva, instalando Ubuntu 18.04.3, Clang-8, libclang-8-dev y make, todo ello mediante la siguiente sentencia:

```
sudo apt install clang-8 libclang-8-dev make
```

Posteriormente, dado que el Makefile para compilar MuCPP utiliza directrices denominando a Clang-version como Clang, Clang++-version como Clang++ y llvm-config-version como llvm-config, se crean los enlaces simbólicos correspondientes mediante las siguientes sentencias:

```
sudo ln -s /usr/bin/clang-8 /usr/bin/clang
sudo ln -s /usr/bin/clang++-8 /usr/bin/clang++
sudo ln -s /usr/bin/llvm-config-8 /usr/bin/llvm-config
```

Una vez realizadas todas estas tareas, se intenta compilar MuCPP, aportando diversos errores, en primer lugar no se encuentran las librerías de Clang, esto se debe a que en el propio Makefile, se define la carpeta de las librerías de llvm-6.0, para ello, lo modificamos pasando de *-I/usr/lib/llvm-6.0/include* a *-I/usr/lib/llvm-8/include*. Gracias a ello ya se encuentran las diferentes librerías de Clang como son Frontend, Serialization, Driver, Tooling, AST, ASTMatchers, entre otras.

Tras esto empezamos a encontrar errores menores, como que el siguiente AST Matcher ya se contempla en el archivo ASTMatchers.h de la librería ASTMatchers de clang 8, concretamente en su línea 640.

```
AST_MATCHER(clang::FunctionDecl, isMain) {
    return Node.isMain();
}
```

Dado que ya se contempla y es factible su utilización, se decide eliminarla del archivo mutationmatchers.cpp de nuestro programa.

Posteriormente se encuentran otros errores en el archivo Mutation.cpp, fácilmente solucionables aplicando los siguientes cambios:

Original	**Cambio**
getLocStart()	getBeginLoc()
getLocEnd()	getEndLoc()

Tabla 53: Tabla de cambios en el fichero Mutation.cpp de MuCPP para actualización a Clang-6.0.

Por último se encuentra una ambigüedad en el uso de *sort(listSources.begin(), listSources.end());*, por ello, se asigna el uso al namespace de std, mediante *std::sort(listSources.begin(), listSources.end());*.

Tras todos estos cambios se compila sin problemas MuCPP y su uso es idéntico, por lo que se puede afirmar que MuCPP ha sido actualizado a Clang-8.

Capítulo 8

Pruebas sobre nuevos operadores

En este capítulo podemos encontrar algunas de las pruebas realizadas a todos y cada uno de los operadores de nueva inclusión en la herramienta MuCPP. Estas pruebas se componen en primer lugar del código de entrada a MuCPP y las salidas (mutantes) que nos devuelve. Los operadores que a los que se han realizado las pruebas pueden observarse en la tabla 54.

Operador
[SOR] Operador de reemplazo de operadores de desplazamiento
[SDL] Eliminación de sentencias
[ODL] Eliminación de operador
[CSD] Eliminación de la palabra clave static
[CSI] Inserción de la palabra clave static

Tabla 54: Tabla operadores no implementados hasta la fecha en MuCPP.

8.1. Pruebas sobre el operador SOR

La entrada para este operador se compone de una función que no devuelve nada en la que se realizan varias operaciones de desplazamiento usando todos y cada uno de los operadores de desplazamiento que admite el operador SOR.

```
1   void funcion(){
2       int a = 2;
3       a = a << 2;
4       a = a >> 2;
5       a <<= 2;
6       a >>= 2;
7   }
```

Este código devuelve un total de cuarenta y cuatro mutantes aunque los que nos interesan son concretamente los cuatro últimos que son listados a continuación. Destacar también que la herramienta tiene asignado el identificador 43 para el operador SOR, el cual estamos probando.

```
The mutant 'm43_1_1_Pruebas_SOR' has been created.
The mutant 'm43_2_1_Pruebas_SOR' has been created.
The mutant 'm43_3_1_Pruebas_SOR' has been created.
The mutant 'm43_4_1_Pruebas_SOR' has been created.
```

A continuación se pueden observar todas y cada una de las modificaciones que realiza cada uno de los mutantes en las diferentes líneas del código fuente aportado anteriormente.

1. El mutante m43_1_1_Pruebas_SOR contiene la siguiente mutación en la línea 5: ***a*** $>>=$ ***2***.
2. El mutante m43_2_1_Pruebas_SOR contiene la siguiente mutación en la línea 6: ***a*** $<<=$ ***2***.
3. El mutante m43_3_1_Pruebas_SOR contiene la siguiente mutación en la línea 3: ***a*** $>>$ ***2***.
4. El mutante m43_4_1_Pruebas_SOR contiene la siguiente mutación en la línea 4: ***a*** $<<$ ***2***.

8.2. Pruebas sobre el operador SDL

La entrada para este operador se compone de una función con cada una de las opciones que muta el operador SDL, sobre el que deseamos realizar las pruebas. El código es el siguiente.

```
1   int funcion(){
2       int a = 0;
3
4       while(a==0){
5           a++;
6       }
7
8       for(int b=0; b<1; b++){
9           a++;
10      }
11
12      if(a==1){
13          a++;
14      }else{
15          a--;
16      }
17
18      switch(a){
19          case 1: a++; break;
20          default: break;
21      }
22
23      return a;
24  }
```

Este código devuelve un total de ciento cinco mutantes aunque los que nos interesan son concretamente aquellos que poseen el identificador 45, ya que MuCPP tiene ese identificador para el operador SDL, el cual estamos probando. En la siguiente lista se pueden observar los mutamtes que nos interesan.

```
The mutant 'm45_1_1_Pruebas_SDL' has been created.
The mutant 'm45_2_1_Pruebas_SDL' has been created.
The mutant 'm45_2_2_Pruebas_SDL' has been created.
The mutant 'm45_3_1_Pruebas_SDL' has been created.
The mutant 'm45_3_2_Pruebas_SDL' has been created.
The mutant 'm45_3_3_Pruebas_SDL' has been created.
The mutant 'm45_4_1_Pruebas_SDL' has been created.
The mutant 'm45_4_2_Pruebas_SDL' has been created.
The mutant 'm45_4_3_Pruebas_SDL' has been created.
The mutant 'm45_5_1_Pruebas_SDL' has been created.
The mutant 'm45_6_1_Pruebas_SDL' has been created.
The mutant 'm45_7_1_Pruebas_SDL' has been created.
The mutant 'm45_8_1_Pruebas_SDL' has been created.
The mutant 'm45_9_1_Pruebas_SDL' has been created.
The mutant 'm45_10_1_Pruebas_SDL' has been created.
The mutant 'm45_11_1_Pruebas_SDL' has been created.
The mutant 'm45_12_1_Pruebas_SDL' has been created.
The mutant 'm45_13_1_Pruebas_SDL' has been created.
```

A continuación se pueden observar todas y cada una de las modificaciones que realiza cada uno de los mutantes en las diferentes líneas del código fuente aportado anteriormente.

1. El mutante m45_1_1_Pruebas_SDL elimina la línea 2.
2. El mutante m45_2_1_Pruebas_SDL contiene la siguiente mutación en la línea 4: ***while(true){*** .
3. El mutante m45_2_2_Pruebas_SDL elimina las líneas de la 4 a la 6.
4. El mutante m45_3_1_Pruebas_SDL elimina las líneas de la 8 a la 10.
5. El mutante m45_3_2_Pruebas_SDL contiene la siguiente mutación en la línea 8: ***for(int b=0; ; b++){***.
6. El mutante m45_3_3_Pruebas_SDL contiene la siguiente mutación en la línea 8: ***for(int b=0; b¡1;){***.
7. El mutante m45_4_1_Pruebas_SDL contiene la siguiente mutación en la línea 12: ***if(true){***.
8. El mutante m45_4_2_Pruebas_SDL contiene la siguiente mutación en la línea 8: ***if(false){***.
9. El mutante m45_4_3_Pruebas_SDL elimina las líneas de la 12 a la 16.
10. El mutante m45_5_1_Pruebas_SDL elimina la línea 20.
11. El mutante m45_6_1_Pruebas_SDL contiene la siguiente mutación en la línea 23: ***return 0;***.
12. El mutante m45_7_1_Pruebas_SDL elimina la línea 5.
13. El mutante m45_8_1_Pruebas_SDL elimina la línea 9.

14. El mutante m45_9_1_Pruebas_SDL elimina la línea 13.

15. El mutante m45_10_1_Pruebas_SDL elimina la línea 15.

16. El mutante m45_11_1_Pruebas_SDL contiene la siguiente mutación en la línea 19: ***case 1: ; break;***.

17. El mutante m45_12_1_Pruebas_SDL contiene la siguiente mutación en la línea 19: ***case 1: a++; ;***.

18. El mutante m45_13_1_Pruebas_SDL contiene la siguiente mutación en la línea 20: ***default: ;***.

8.3. Pruebas sobre el operador ODL

La entrada para este operador se compone de una función que no devuelve nada en la que se realizan varias operaciones con operadores binarios con y sin asignación usando todos y cada uno de los operadores binarios que admite el operador ODL.

```
1   void funcion(){
2       int a = 0;
3       a = a + 2;
4       a = a - 2;
5       a = a * 2;
6       a = a / 2;
7       a = a % 2;
8       a = a < 2;
9       a = a > 2;
10      a = a >= 2;
11      a = a >= 2;
12      a = a == 2;
13      a = a != 2;
14      a = a && a;
15      a = a & 2;
16      a = a || a;
17      a = a | 2;
18      a = a ^ 2;
19      a = a << 2;
20      a = a >> 2;
21      a += 2;
22      a -= 2;
23      a *= 2;
24      a /= 2;
25      a %= 2;
26      a <<= 2;
27      a >>= 2;
28      a &= 2;
29      a |= 2;
30      a ^= 2;
31  }
```

Este código devuelve un total de cuatrocientos tres mutantes aunque los que nos interesan son concretamente los cuarenta y dos mutantes que tienen asignados el identificador 44, ya que es el identificador asignado para el operador ODL, el cual estamos probando. Los mutantes se pueden observar en la siguiente lista.

```
The mutant 'm44_1_1_Pruebas_ODL' has been created.
The mutant 'm44_2_1_Pruebas_ODL' has been created.
The mutant 'm44_3_1_Pruebas_ODL' has been created.
The mutant 'm44_4_1_Pruebas_ODL' has been created.
The mutant 'm44_5_1_Pruebas_ODL' has been created.
The mutant 'm44_6_1_Pruebas_ODL' has been created.
The mutant 'm44_7_1_Pruebas_ODL' has been created.
The mutant 'm44_8_1_Pruebas_ODL' has been created.
The mutant 'm44_9_1_Pruebas_ODL' has been created.
The mutant 'm44_10_1_Pruebas_ODL' has been created.
The mutant 'm44_11_1_Pruebas_ODL' has been created.
The mutant 'm44_11_2_Pruebas_ODL' has been created.
The mutant 'm44_12_1_Pruebas_ODL' has been created.
The mutant 'm44_12_2_Pruebas_ODL' has been created.
The mutant 'm44_13_1_Pruebas_ODL' has been created.
The mutant 'm44_13_2_Pruebas_ODL' has been created.
The mutant 'm44_14_1_Pruebas_ODL' has been created.
The mutant 'm44_14_2_Pruebas_ODL' has been created.
The mutant 'm44_15_1_Pruebas_ODL' has been created.
The mutant 'm44_15_2_Pruebas_ODL' has been created.
The mutant 'm44_16_1_Pruebas_ODL' has been created.
The mutant 'm44_16_2_Pruebas_ODL' has been created.
The mutant 'm44_17_1_Pruebas_ODL' has been created.
The mutant 'm44_17_2_Pruebas_ODL' has been created.
The mutant 'm44_18_1_Pruebas_ODL' has been created.
The mutant 'm44_19_1_Pruebas_ODL' has been created.
The mutant 'm44_20_1_Pruebas_ODL' has been created.
The mutant 'm44_21_1_Pruebas_ODL' has been created.
The mutant 'm44_22_1_Pruebas_ODL' has been created.
The mutant 'm44_22_2_Pruebas_ODL' has been created.
The mutant 'm44_23_1_Pruebas_ODL' has been created.
The mutant 'm44_23_2_Pruebas_ODL' has been created.
The mutant 'm44_24_1_Pruebas_ODL' has been created.
The mutant 'm44_24_2_Pruebas_ODL' has been created.
The mutant 'm44_25_1_Pruebas_ODL' has been created.
The mutant 'm44_25_2_Pruebas_ODL' has been created.
The mutant 'm44_26_1_Pruebas_ODL' has been created.
The mutant 'm44_26_2_Pruebas_ODL' has been created.
The mutant 'm44_27_1_Pruebas_ODL' has been created.
The mutant 'm44_27_2_Pruebas_ODL' has been created.
The mutant 'm44_28_1_Pruebas_ODL' has been created.
The mutant 'm44_28_2_Pruebas_ODL' has been created.
```

A continuación se pueden observar todas y cada una de las modificaciones que realiza cada uno de los mutantes en las diferentes líneas del código fuente aportado anteriormente. Estos mutantes tienen un orden diferente a los de las diferentes líneas de código fuente que se le han sido aportadas. Esto se debe a que el AST de Clang realiza unas operaciones antes que otras.

1. El mutante m44_1_1_Pruebas_ODL contiene la siguiente mutación en la línea 21:

a = 2;.

2. El mutante m44_2_1_Pruebas_ODL contiene la siguiente mutación en la línea 22: ***a = 2;***.
3. El mutante m44_3_1_Pruebas_ODL contiene la siguiente mutación en la línea 23: ***a = 2;***.
4. El mutante m44_4_1_Pruebas_ODL contiene la siguiente mutación en la línea 24: ***a = 2;***.
5. El mutante m44_5_1_Pruebas_ODL contiene la siguiente mutación en la línea 25: ***a = 2;***.
6. El mutante m44_6_1_Pruebas_ODL contiene la siguiente mutación en la línea 26: ***a = 2;***.
7. El mutante m44_7_1_Pruebas_ODL contiene la siguiente mutación en la línea 27: ***a = 2;***.
8. El mutante m44_8_1_Pruebas_ODL contiene la siguiente mutación en la línea 28: ***a = 2***.
9. El mutante m44_9_1_Pruebas_ODL contiene la siguiente mutación en la línea 29: ***a = 2;***.
10. El mutante m44_10_1_Pruebas_ODL contiene la siguiente mutación en la línea 30: ***a = 2;***.
11. El mutante m44_11_1_Pruebas_ODL contiene la siguiente mutación en la línea 3: ***a = 2;***.
12. El mutante m44_11_2_Pruebas_ODL contiene la siguiente mutación en la línea 3: ***a = a;***.
13. El mutante m44_12_1_Pruebas_ODL contiene la siguiente mutación en la línea 4: ***a = 2;***.
14. El mutante m44_12_2_Pruebas_ODL contiene la siguiente mutación en la línea 4: ***a = a;***.
15. El mutante m44_13_1_Pruebas_ODL contiene la siguiente mutación en la línea 5: ***a = 2;***.
16. El mutante m44_13_2_Pruebas_ODL contiene la siguiente mutación en la línea 5: ***a = a;***.
17. El mutante m44_14_1_Pruebas_ODL contiene la siguiente mutación en la línea 6: ***a = 2;***.
18. El mutante m44_14_2_Pruebas_ODL contiene la siguiente mutación en la línea 6: ***a = a;***.
19. El mutante m44_15_1_Pruebas_ODL contiene la siguiente mutación en la línea 7: ***a = 2;***.

20. El mutante m44_15_2_Pruebas_ODL contiene la siguiente mutación en la línea 7: ***a = a;***.
21. El mutante m44_16_1_Pruebas_ODL contiene la siguiente mutación en la línea 8: ***a = 2;***.
22. El mutante m44_16_2_Pruebas_ODL contiene la siguiente mutación en la línea 8: ***a = a;***.
23. El mutante m44_17_1_Pruebas_ODL contiene la siguiente mutación en la línea 9: ***a = 2;***.
24. El mutante m44_17_2_Pruebas_ODL contiene la siguiente mutación en la línea 9: ***a = a;***.
25. El mutante m44_18_1_Pruebas_ODL contiene la siguiente mutación en la línea 10: ***a = 2;***.
26. El mutante m44_18_2_Pruebas_ODL contiene la siguiente mutación en la línea 10: ***a = a;***.
27. El mutante m44_19_1_Pruebas_ODL contiene la siguiente mutación en la línea 11: ***a = 2;***.
28. El mutante m44_19_2_Pruebas_ODL contiene la siguiente mutación en la línea 11: ***a = a;***.
29. El mutante m44_20_1_Pruebas_ODL contiene la siguiente mutación en la línea 12: ***a = 2;***.
30. El mutante m44_20_2_Pruebas_ODL contiene la siguiente mutación en la línea 12: ***a = a;***.
31. El mutante m44_21_1_Pruebas_ODL contiene la siguiente mutación en la línea 13: ***a = 2;***.
32. El mutante m44_21_2_Pruebas_ODL contiene la siguiente mutación en la línea 13: ***a = a;***.
33. El mutante m44_22_1_Pruebas_ODL contiene la siguiente mutación en la línea 14: ***a = 2;***.
34. El mutante m44_22_2_Pruebas_ODL contiene la siguiente mutación en la línea 14: ***a = a;***.
35. El mutante m44_23_1_Pruebas_ODL contiene la siguiente mutación en la línea 15: ***a = 2;***.
36. El mutante m44_23_2_Pruebas_ODL contiene la siguiente mutación en la línea 15: ***a = a;***.
37. El mutante m44_24_1_Pruebas_ODL contiene la siguiente mutación en la línea 16: ***a = 2;***.
38. El mutante m44_24_2_Pruebas_ODL contiene la siguiente mutación en la línea 16: ***a = a;***.

39. El mutante m44_25_1_Pruebas_ODL contiene la siguiente mutación en la línea 17: ***a = 2;***.
40. El mutante m44_25_2_Pruebas_ODL contiene la siguiente mutación en la línea 17: ***a = a;***.
41. El mutante m44_26_1_Pruebas_ODL contiene la siguiente mutación en la línea 18: ***a = 2;***.
42. El mutante m44_26_2_Pruebas_ODL contiene la siguiente mutación en la línea 18: ***a = a;***.
43. El mutante m44_27_1_Pruebas_ODL contiene la siguiente mutación en la línea 19: ***a = 2;***.
44. El mutante m44_27_2_Pruebas_ODL contiene la siguiente mutación en la línea 19: ***a = a;***.
45. El mutante m44_28_1_Pruebas_ODL contiene la siguiente mutación en la línea 20: ***a = 2;***.
46. El mutante m44_28_2_Pruebas_ODL contiene la siguiente mutación en la línea 20: ***a = a;***.

8.4. Pruebas sobre el operador CSD

La entrada para este operador se compone de una clase en la que se crea una unica variable static para que MuCPP utilice el operador CSD el cual está siendo probado.

```
1   class clase{
2       inline static bool booleano = true;
3   };
```

Este código devuelve un solo mutante que tienen asignados el identificador 46, ya que es el identificador asignado para el operador CSD, el cual estamos probando.

```
The mutant 'm46_1_1_Pruebas_CSD' has been created.
```

A continuación se puede observar la modificación que realiza el mutante creado.

1. El mutante m46_1_1_Pruebas_CSD contiene la siguiente mutación en la línea 2: ***bool booleano = true;***.

8.5. Pruebas sobre el operador CSI

La entrada para este operador se compone de una clase en la que se crea una unica variable static para que MuCPP utilice el operador CSI el cual está siendo probado.

```
1   class clase{
2       bool booleano = true;
3   };
```

Este código devuelve un solo mutante que tienen asignados el identificador 47, ya que es el identificador asignado para el operador CSI, el cual estamos probando.

```
The mutant 'm47_1_1_Pruebas_CSI' has been created.
```

A continuación se puede observar la modificación que realiza el mutante creado.

1. El mutante m47_1_1_Pruebas_CSI contiene la siguiente mutación en la línea 2: ***inline static bool booleano = true;***.

Capítulo 9

Conclusiones y trabajo futuro

En el presente capítulo se expondrán las conclusiones a las que se llega tras la actualización de la herramienta de prueba de mutaciones MuCPP y el trabajo futuro a realizar sobre ella.

9.1. Conclusiones

Tras la realización del presente proyecto se llega a la conclusión de que MuCPP ha sido actualizado para su puesta en funcionamiento en la industria actual, teniendo una gran cantidad de operadores de mutación con los que realizar prueba de mutaciones y poder realizar pruebas sobre clases o métodos concretos en un entorno totalmente actualizado a Clang-8, la ultima versión del conocido compilador.

Destacar que se ha aprendido mucho sobre la prueba de mutaciones, un campo en el que no se había trabajado hasta el momento, conociendo herramientas de este campo y actualizando herramientas que han sido desarrolladas por otro desarrollador, cosa nada sencilla teniendo en cuenta las diferencias que existen habitualmente entre desarrolladores.

Añadir también que la prueba de mutaciones es un campo que parece muy interesante dado que realiza cambios en el código de forma automática además de utilizar los AST que se aprenden en diferentes asignaturas de la especialidad de computación.

9.2. Trabajo futuro

En el trabajo futuro se encuentran diversas actuaciones sobre la herramienta de prueba de mutaciones MuCPP, los cuales son:

- Creación de una interfaz gráfica para la herramienta MuCPP, mediante la que el usuario final pueda interaccionar y no deba usar la línea de comandos en ningún momento, mejorando así la usabilidad de la herramienta.

- Realización de artículos de investigación sobre la prueba de mutaciones y MuCPP, mediante los que dar a conocer las mejoras implementadas en el presente proyecto.

- Investigación sobre operadores no funcionales para la herramienta MuCPP y su inclusión en esta, haciendo la herramienta más sofisticada y con un gran número de operadores con los que mejorar el desarrollo del software.
- Puesta en marcha de la herramienta en entornos de Industria 4.0, haciendo un análisis de su funcionamiento. Este trabajo futuro se crea con la intención de implantar mejoras en entornos industriales de la provincia o alrededores dando a conocer la herramienta.

Capítulo 10

Glosario y definiciones

10.1. Glosario

ADS: Arithmetic Operator Deletion.

AIS: Arithmetic Operator Insertion.

AIU: Arithmetic Operator Insertion.

ARB: Arithmetic Operator Replacement.

ARS: Arithmetic Operator Replacement.

ARU: Arithmetic Operator Replacement.

ASR: Short-Cut Assignment Operator Replacement.

AST: Árbol de sintaxis abstracta.

CCA: Copy constructor and assignment operator overloading deletion.

CDC: Default constructor creation.

CDD: Destructor method deletion.

CDL: Constant DeLetion.

CID: Member variable initialization deletion.

COD: Conditional Operator Deletion.

COI: Conditional Operator Insertion.

COR: Conditional Operator Replacement.

CTD: this keyword deletion.

CTI: this keyword insertion.

EAM: Acessor method change.

EHC: Exception handling change.

EHR: Exception handler removal.

EMM: Modifier method change.

EOA: Reference assignment and content assignment replacement.

EOC: Reference comparison and content comparison replacement.

IHD: Hiding variable deletion.

IHI: Hiding variable insertion.

IMR: Multiple inheritance replacement.

IOD: Overriding method deletion.

IOP: Overriding method calling position change.

IOR: Overriding method rename.

IPC: Explicit call of a parent's constructor deletion.

ISD: Base keyword deletion.

ISI: Base keyword insertion.

ISO : International Organization for Standarization .

JSD: Static modifier deletion.

JSI: Static modifier insertion.

LLVM: Low-Level Virtual Machine.

LOR: Logical Operator Replacement.

MCI: Member call from another inherited class.

MCO: Member call from another object.

OAN: Argument number change.

OAO: Argument order change.

ODL: Operator DeLetion.

OMD: Overloading method deletion.

OMR: Overloading method contents replace.

PCC: Cast type change.

PCD: Type cast operator deletion.

PCI: Type cast operator insertion.

PMD: Member variable declaration with parent class type.

PNC: New method call with child class type.

PPD: Parameter variable declaration with child class type.

PRV: Reference assignment with other comparable variable.

PVI: virtual modifier insertion.

ROR: Relational Operator Replacement.

SDL: Statement DeLetion.

SOR: Shift Operator Replacement.

TFG: Trabajo Fin de Grado.

UCA: Universidad de Cádiz.

VDL: Variable DeLetion.

10.2. Definiciones

Prueba de mutaciones. La prueba de mutaciones es una técnica de caja blanca que consiste en la introducción de pequeños fallos en el código fuente de un determinado programa y observar si dicho cambio es identificado o no por las pruebas realizadas por el desarrollador.

Operadores de mutación. Reglas de transformación predefinidas que simulan fallos de programación habituales que son inyectados en el código fuente mediante la prueba de mutaciones.

Árbol de Sintaxis Abstracta. Estructura de datos ampliamente usada en compiladores para representar el código de un programa. Usualmente es el resultado de la fase de análisis sintáctico de un compilador.

AST Matchers Predicados para la búsqueda y acceso a los nodos del AST. Son creados mediante llamadas a funciones que permiten crear un árbol de patrones (matchers), donde los matchers internos se usan para hacer la selección más específica.

Clang: Frontend de compilador para los lenguajes de programación C, C++, Objective-C y Objective-C++.

LLVM: Infraestructura para desarrollar compiladores, escrita a su vez en el lenguaje de programación C++, que está diseñada para optimizar el tiempo de compilación, el tiempo de enlazado, el tiempo de ejecución y el "tiempo ocioso" en cualquier lenguaje de programación que el usuario quiera definir.

Apéndice A

MuCPP: manual de instalación y uso

En este anexo se podrá encontrar un manuel de instalación de la herramienta de prueba de mutaciones MuCPP además de una explicación de como utilizar la herramienta.

A.1. Instalación de MuCPP

En esta sección se explica como instalar la última versión disponible de MuCPP, la cual puede ser obtenida como anexo a este documento o desde la web oficial de la herramienta en [17].

La instalación es muy sencilla, solo será necesario poseer un dispositivo con Ubuntu 18.04.3, además de esto, este sistema deberá poseer Clang-8, libclang-8-dev y git, los cuales podrán ser instalados mediante la siguiente sentencia en la terminal del sistema:

```
sudo apt install clang-8 libclang-8-dev git
```

Tras esto crearemos diferentes enlaces simbólicos para que la herramienta pueda funcionar sin ningún problema, para ello solo será necesario ejecutar las siguientes sentencias:

```
sudo ln -s /usr/bin/clang-8 /usr/bin/clang
sudo ln -s /usr/bin/clang++-8 /usr/bin/clang++
sudo ln -s /usr/bin/llvm-config-8 /usr/bin/llvm-config
```

Posteriormente se deberá configurar git, mediante las siguientes sentencias, las cuales deberán ser modificadas por el lector:

```
git config --global user.name "Nombre"
git config --global user.email "Correo"
```

Tras realizar estas acciones pasaremos a instalar la herramienta, para ello se deberá acceder a la carpeta MuCPP de la carpeta de archivos anexos:

```
cd MuCPP
```

Posteriormente se le darán permisos de ejecución a la herramienta:

```
ls -l MuCPP
chmod +x MuCPP
```

Por último se copiará la herramienta a la carpeta /usr/bin del sistema operativo:

```
sudo cp MuCPP /usr/bin
```

A.2. Funcionamiento de MuCPP

El funcionamiento de MuCPP es muy sencillo, desarrollándose en siete campos bien diferenciados, los cuales serán descritos a continuación. Toda esta información se ha obtenido de [17] y en ella se puede encontrar información complementaria sobre el uso de la herramienta o ejemplos de funcionamiento.

A.2.1. Notas previas

- Para obtener información sobre las opciones disponibles, códigos de operador, etc, debe escribir:

  ```
  MuCPP --help
  ```

- Para eliminar los mutantes creados previamente en el sistema bajo prueba y comenzar desde una copia limpia, use la opción:

  ```
  MuCPP clean
  ```

- En los ejemplos que se muestran para cada opción, se supone la existencia de una base de datos de compilación. De lo contrario, " − " debe incluirse al final de los comandos (como se explica en la sección Requisitos").

- Las opciones cuentan, analizan y aplican todas: si no se proporciona ningún archivo fuente, se procesarán los archivos en el directorio raíz (en este caso, coloque los archivos fuente de prueba en un directorio de prueba como un subdirectorio dentro del directorio raíz).

- Solo se permiten los archivos con extensión " c " , " cc ", " cxx " o " cpp ". Los archivos de encabezado serán mutados para ser incluidos en un archivo de código fuente. Si un mismo archivo de encabezado está presente en más de un archivo fuente, la herramienta evitará la creación de mutantes duplicados. Los archivos fuente proporcionados se ordenan a través de " std::sort " para que todas las ejecuciones con los mismos archivos siempre reporten los mismos mutantes.

A.2.2. Count

Mediante esta sentencia se calcula el total de mutantes en los archivos suministrados.

```
MuCPP count [file1.cpp file2.cpp...]
```

A.2.3. Analyze

Mediante esta sentencia se muestra un informe con los mutantes posibles para cada operador:

```
MuCPP analyze [file1.cpp file2.cpp ...]
```

Los informes se muestra en la pantalla, pero también se guarda en el directorio denominado " reports_analyze ", siendo estos:

- Un informe por archivo fuente analizado, con el nombre "analyze_file.txt " (donde *archivo* es el nombre del archivo de código fuente sin extensión).
- Un informe global que calcula los mutantes de todos los archivos de código fuente en su conjunto, con el nombre "analyze_MuCPP_global.txt".

La salida por pantalla es una lista con la siguiente información para cada operador de mutación:
"Operador ubicaciones atributos"
donde:

- **Operador**. Es el código del operador que corresponde.
- **Ubicaciones**. Es el número de ubicaciones donde es posible introducir una mutación de ese operador.
- **Atributos**. Número de mutaciones posibles a insertar en cada ubicación (*). Por ejemplo, si "atributos" muestra "2", eso significa que se pueden introducir dos mutantes diferentes por ubicación de mutación.

(*) Hay operadores de mutación cuyo número de atributos no es fijo, sino variable. En otras palabras, el número de mutantes para insertar en una ubicación concreta depende completamente de la ubicación dentro del código. En este caso, el atributo se marca como "1" y cada mutante producido se agrega al contador de ubicación.

A.2.4. Applyall

Mediante la siguiente sentencia se generarán todos los mutantes.

```
MuCPP applyall [archivo1.cpp archivo2.cpp ...]
```

Los mutantes se crean como ramas en el sistema de control de versiones git. Las ramas contienen una copia exacta del programa original, excepto del archivo o archivos mutados.
Formato para el nombre de mutantes:
"m + código de operador + _ + orden de ubicación + _ + atributo + _ + nombre del archivo"

Dónde:

- **Código**. Es el código del operador que corresponde.
- **Ubicación**. Es el número de ubicación del operador en el código.
- **Atributo**. Numero de la mutación a insertar en esa ubicación.

- **Nombre del archivo**. Corresponde al nombre del archivo sin el punto ni su terminación.

Notas:

- Cuando se suministra applyall, las ramas existentes se eliminan previamente.
- Los mutantes se generan a medida que las ubicaciones para mutar se encuentran en el recorrido del código.

A.2.5. Apply

Generación de un solo mutante identificado por operador, ubicación, atributo y archivo, haciéndolo mediante la siguiente sentencia:

```
MuCPP apply operador localización atributo [file1.cpp file2.cpp ...]
```

A.2.6. Run

Ejecución del conjunto de pruebas en un programa. El programa original se ejecuta cuando se suministran mutantes. En caso de que se proporcionen algunos mutantes, el conjunto de pruebas se ejecutará contra esos mutantes.

```
MuCPP run test_directory [mutant1 mutant2 ...]
```

Se genera un archivo llamado "tests_output.txt", donde cada línea es el resultado de un caso de prueba, siendo cero para una prueba satisfactoria y uno para una no satisfactoria. Si el mutante no es válido se mostrará un dos.

Si se han medido los tiempos de ejecución del conjunto de pruebas, se creará un archivo "times_output.txt", donde cada línea representa el tiempo empleado por cada caso de prueba.

Nota: Los archivos resultantes ("tests_output.txt" y "times_output.txt") se crearán en el directorio raíz y se versionarán en git.

A.2.7. Compare

Esta opción compara entre los resultados de la ejecución del conjunto de pruebas con el programa original y los resultados para los mutantes suministrados. Si no se suministran mutantes, se ejecutarán todos los mutantes existentes.

```
MuCPP compare test_directory [mutant1 mutant2 ...]
```

Este comando muestra una línea por mutante, donde para cada caso de prueba se muestra un cero para el mismo resultado, un uno para un resultado diferente y un dos en caso de un mutante no válido.

Si se midieron los tiempos de ejecución del conjunto de pruebas, se mostrarán junto a los resultados (se utiliza una " T " para separar los resultados y los tiempos).

Los resultados de la ejecución del conjunto de pruebas para los mutantes seleccionados también se recopilarán en un archivo llamado "comparsion_results.txt" en el directorio raíz cuando finalice la ejecución.

Apéndice B

Operadores individuales: instalación y uso

En este anexo encontrará un manual de instalación de los operadores individuales incluidos en las carpetas adjuntas y un manual de uso de estos operadores.

B.1. Instalación de los operadores individuales

En esta sección se explica como instalar la última versión disponible de MuCPP, la cual puede ser obtenida como anexo a este documento.

La instalación es muy sencilla, solo será necesario poseer un dispositivo con Ubuntu 18.04.3, además de esto, este sistema deberá poseer Clang-8, libclang-8-dev y git, los cuales podrán ser instalados mediante la siguiente sentencia en la terminal del sistema:

```
sudo apt install clang-8 libclang-8-dev git
```

Tras esto crearemos diferentes enlaces simbólicos para que la herramienta pueda funcionar sin ningún problema, para ello solo será necesario ejecutar las siguientes sentencias:

```
sudo ln -s /usr/bin/clang-8 /usr/bin/clang
sudo ln -s /usr/bin/clang++-8 /usr/bin/clang++
sudo ln -s /usr/bin/llvm-config-8 /usr/bin/llvm-config
```

Posteriormente se deberá configurar git, mediante las siguientes sentencias, las cuales deberán ser modificadas por el lector:

```
git config --global user.name "Nombre"
git config --global user.email "Correo"
```

Tras realizar estas acciones pasaremos a instalar el operador, para ello se deberá acceder a la carpeta con el nombre del operador deseado de la carpeta de operadores de los archivos anexos:

```
cd operadores\ individuales
cd nombre_operador_deseado
```

Tras esto, se deberá compilar el operador mediante la sentencia

```
make
```

Posteriormente se le darán permisos de ejecución al operador:

```
ls -l nombre_operador_deseado
chmod +x nombre_operador_deseado
```

Por último se copiará la herramienta a la carpeta /usr/bin del sistema operativo:

```
sudo cp nombre_operador_deseado /usr/bin
```

B.2. Uso del operador

Para realizar el uso del operador deseado solo se ejecutará la siguiente sentencia la terminal en la carpeta del archivo con el código fuente que deseamos mutar, aportando posteriormente todos los mutantes generador por el operador uno tras otro, informando de la linea y columna en la que se encuentra cada uno de ellos antes del código mutado.

```
nombre_operador_deseado nombre_archivo_codigo_fuente
```

Si lo desea, para una sencilla compilación y prueba con las pruebas realizadas en el capítulo 8, solo debe acceder a la carpeta del operador deseado y ejecutar la siguiente sentencia:

```
sh make.sh
```

Recordar que las pruebas también han sido aportadas para cada operador en la carpeta Pruebas.

Bibliografía

[1] Grupo UCASE de Ingeniería del Software - TIC025. `https://tic025.uca.es`. Ultimo acceso: 08-07-2019.

[2] C. Sandler G. J. Myers and T. Badgett. *The Art of Software Testing, 3rd Edition. (3rd ed.).* 2011.

[3] Jie Zhang Yue Jia Yves Le Traon Mike Papadakis, Marinos Kintis and Mark Harman. *Mutation Testing Advances: An Analysis and Survey.* Elsevier, 2017.

[4] P. Delgado-Pérez et al. *Assessment of class mutation operators for C++ with the MuCPP mutation system,.* Information and Software Technology, vol. 81, pp. 169-184, 2017.

[5] J. Lajoie S. B. Lippman and B. E. Moo. *C++ Primer, Fifth Edition. (5th ed.).* Addison-Wesley Professional, 2012.

[6] F. Palomo Lozano et al. *Fundamentos De C++. (2^a corr. y aum. ed.).* Universidad de Cádiz. Servicio de Publicaciones, 2016.

[7] Tiobe index. `https://www.tiobe.com/tiobe-index/`. Ultimo acceso: 17-08-2019.

[8] A. Prieto Espinosa and B. Prieto Campos. *Conceptos De Informática.* 2005.

[9] R. S. Pressman. *Ingeniería Del Software: Un Enfoque Práctico. (7^a ed.) Mexico D.F: McGraw-Hill.* 2010.

[10] G. J. Myers et al. *The Art of Software Testing. (2nd ed.) Hoboken: John Wiley Sons.* 2004.

[11] R. Lipton. *Fault diagnosis of computer programs, Student Report, Carnegie Mellon University.* 1971.

[12] R. G. Hamlet. *Testing Programs with the Aid of a Compiler, IEEE Transactions on Software Engineering, vol. SE-3, (4), pp. 279-290.* 1977.

[13] R. J. Lipton R. A. DeMillo and F. G. Sayward. *Hints on Test Data Selection: Help for the Practicing Programmer, Computer, vol. 11, (4), pp. 34-41.* 1978.

[14] The LLVM Compiler Infrastructure. `http://llvm.org`. Ultimo acceso: 09-07-2019.

[15] Pedro delgado-pérez, inmaculada medina-bulo and juan josé domínguez-jiménez, "generación de mutantes válidos en el lenguaje de programación c++,". `https://tinyurl.com/y6ft5agk`. Ultimo acceso: 22-08-2019.

[16] Clang: a c language family frontend for llvm. `https://clang.llvm.org`. Ultimo acceso: 22-08-2019.

[17] MuCPP Mutation Tool. `https://ucase.uca.es/MuCPP/`. Ultimo acceso: 08-07-2019.

[18] Parasoft insure++. `http://www.parasoft.com/products/insure`. Ultimo acceso: 22-08-2019.

[19] A. Derezinska and K. Kowalski. *Object-oriented mutation applied in common intermediate language programs originated from C#.* IEEE, 2011, doi:10.1109/ICSTW.2011.54.

[20] Plextest, itregister. `http://www.itregister.com.au/products/plextest`. Ultimo acceso: 22-08-2019.

[21] M. Hampton and S. Petithomme. *Leveraging a commercial mutation analysis tool for research.* 2007, DOI: 10.1109/TAICPART.2007.4344125.

[22] M. Kusano and C. Wang. *CCmutator: A mutation generator for concurrency constructs in multithreaded C/C++ applications.* 2013, DOI: 10.1109/ASE.2013.6693142.

[23] mujava, jeff offutt and nan li. `https://cs.gmu.edu/~offutt/mujava/`. Ultimo acceso: 22-08-2019.

[24] Pit, real world mutation testing. `http://pitest.org`. Ultimo acceso: 22-08-2019.

[25] The major mutation framework. `http://mutation-testing.org`. Ultimo acceso: 22-08-2019.

[26] Jester - the junit test tester. `http://jester.sourceforge.net`. Ultimo acceso: 22-08-2019.

[27] Javalanche. `http://www.st.cs.uni-saarland.de/mutation/`. Ultimo acceso: 22-08-2019.

[28] Testooj. `https://alarcos.esi.uclm.es/testooj3/`. Ultimo acceso: 22-08-2019.

[29] Jumble. `http://jumble.sourceforge.net`. Ultimo acceso: 22-08-2019.

[30] Yu-Seung Ma and Jeff Offutt. *Description of muJava's Method-level Mutation Operators.* in 2011.

[31] Yu-Seung Ma Jeff Offutt and Yong-Rae Kwon. *The Class-Level Mutants of MuJava.* in 2006.

[32] R. Just. *The major mutation framework: Efficient and scalable mutation analysis for java.* In Proceedings of the 2014 international symposium on software testing and analysis, pages 433–436. ACM, 2014.

[33] Jeff Offutt Lin Deng and Nan Li. *Empirical evaluation of the statement deletion mutation operator.* In 6th IEEE International Conference on Software Testing, Verification, and Validation (ICST 2013), pages 80–93, Luxembourg, March 2013.

[34] J. Offutt M. E. Delamaro and P. Ammann. *Designing deletion mutation operators,.* in 2014, DOI: 10.1109/ICST.2014.12.

[35] F. Mariya and D. Barkhas. *A comparative analysis of mutation testing tools for java.* in 2016, DOI: 10.1109/EWDTS.2016.7807636.

[36] A. Márki and B. Lindström. *Mutation tools for java.* in 2017, DOI: 10.1145/3019612.3019825.

[37] Pedro Delgado Pérez. *TESIS DOCTORAL: Evolutionary Mutation Testing in Object-Oriented Environments.* in 2017.

[38] Pedro Delgado Pérez. *Trabajo de Investigación: Definición de operadores de mutación para el lenguaje C++.* in 2012.

[39] C++ data types. `https://www.geeksforgeeks.org/c-data-types/`. Ultimo acceso: 31-08-2019.